Jupyter on Steroids

ABSTRACT

Despite the plethora of recently proposed tools and frameworks, common data analysis tasks such as data retrieval, curation and visualization remain laborious and non-trivial. The limited flexibility of existing frameworks, pushes code-literate data analysts to interactive notebooks such as Jupyter.

In this work, we argue that interactive notebooks are suboptimal for commonly performed tasks, with regards to their ease of use and interactivity. We propose ViDeTTe, a framework designed to facilitate data analysis through the utilization of a template language. We implement a sample data analysis work flow and present the benefits of using ViDeTTe instead of commonly used imperative language in iPython and Jupyter notebooks.

1. INTRODUCTION

User-friendly data analysis tools and frameworks often provide limited flexibility, as they usually focus on a predetermined set of use/analysis cases or a small fraction of the typically large data analysis pipeline. This lack of flexibility often pushes code-literate analysts towards the use of interactive notebooks such as Jupyter.

Interactive notebooks allow the use of popular, high-level and highly expressive imperative languages, such as Python, for analyzing data and composing the results into an easily readable notebook-like interface. Due to the wide popularity of such languages, there is also a huge collection of third-party libraries that can be used by data scientists as building blocks of a much bigger analytical process. Furthermore, the web environment of notebooks enables collaboration between data scientists, since it allows them to directly interact with the user interface in order to develop and run code, process data, generate visualizations, and lastly, compose their findings into an interactive (and re-runnable) report-like page, that contains code, visualizations and textual description of the analysis.

However as we show in this work, interactive notebooks are still suboptimal with regard to ease of use and interactivity. Setting up notebook environments and dependencies, obtaining and combining data and generating the respective visualizations, requires technical knowledge that often exceeds the skill-set of a typical data scientist. Lastly, while such notebooks support the generation of interactive visualizations, this interactivity is not an integral part of the data analysis process.

We address these issues, by extending interactive notebooks with a template language called ViDeTTe. The main contributions of this extension are:

- *Declarative semantics:* ViDeTTe implements formal declarative *Model-View-View-Model* (MVVM) semantics.

  Fill in why this is a good thing. I have no idea.

- *Expressive template language:* Prior database work, treats a page as a database view. Building on that, our template language goes beyond SQL query and view definition in both style and fundamental expressiveness. It is a mixture of query as well as web templating language that works on ordered (arrays) and semi-ordered (JSON) data.

- We allow in-line declarative code directly in JSON...

In this work, we demonstrate the use of ViDeTTe via a walkthrough example. Specifically, we want to use website access data to plot an access count over time histogram. We also want to plot the recorded user demographics (with focus on age groups). We then want to have the ability to interact with the histogram plot and select a time region. This action should automatically update the second plot with the user demographics in the selected time window.

Without loss of generality, we assume a Jupyter server, where the analysts develop their notebooks and a different database server where data is stored. To retrieve the entirety of the required data, we have to query two different databases and join the returned JSON files. Figure ?? shows how our databases are organized. Our fictional analyst will perform the following high-level tasks:

- Data retrieval from remote databases.
- Data curation: Join data and prepare for visualization.
- Data visualization.

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

The remainder of this paper is organized as follows: Sections 2 – 4 present a direct comparison of using ViDeTTe and an imperative language such as Python in order to complete the tasks of our example. Throughout these sections, we demonstrate some of the main contributions of ViDeTTe. Section 5 provides further discussion regarding our proposed extension and presents other useful aspects of it not used in our walkthrough. Finally, Section 6 concludes the paper.

2. DATA RETRIEVAL

> *It might be a good idea to merge these 3 sections into 1 called "Walkthrough example" or something like that. Depends on how large these sections become.*

Data retrieval is one aspect where ViDeTTe shines. The analyst is given the ability to write clean and readable database queries (?? - Is there a better name - ??) *Costas: yep that's how we call them...* in order to retrieve data from the remote server. In a Python implementation, analysts need to perform a series of tasks before being able to retrieve data. Some of these tasks potentially fall way beyond the skill set of the average data analyst.

Costas: We should also say that we simplify file imports. For cases, when the data analysts wants to use csv or json files from their local computer, they can use an import button, which is part of our UI, to upload them to the notebook server. After that step is completed we trigger a function that automatically converts this file to JSON and we assign it to a ViDeTTe variable, so that it can be used in the notebook. Currently, this task requires ssh-ing or scp-ing the files to the notebook server, and potentially moving files to the directive/file-system that is seen by the notebook, which also requires system administrator-linux knowledge

First, our analyst needs to install and configure database drivers. This step will either introduce a dependency between the analyst and some system administrator, or the analyst will need to have the required access rights in order to perform the installation. The later can be a security concern or can lead to corrupted systems if not performed correctly. The complexity of this step increases as the number of different database systems, our analyst wants to access, increases. More specifically, for cases when an analyst needs to access data stored in a MongoDB and a MySQL database, two drivers will need to be installed.

Once the system is configured, the analyst then needs to read lengthy documentation documents in order to properly issue queries to the databases via imperative code, then consume the results, and potentially convert them into a format that assists in data processing. Lastly, the analyst will have to convert the format of the data again in order to feed them into the respective visualization library.

Finally, when implemented in a Jupyter notebook, the analyst's credentials for the database server might lie on plain sight to anyone who has access to the notebook, disrupting the valuable ease of results communication through notebooks.

ViDeTTe addresses each one of the aforementioned imperative code issues efficiently: The analyst generates a configuration file, containing information required for establishing a successful connection with the respective database systems. Figure **??** shows an example of the configuration file that is used for connecting to a postgres database which contains the data that will by used in our analysis. The configuration file must contain the type of the database system, the host name, port and the credentials that will be used for obtaining a connection. Additionally, it contains the database tables that will be accessible by queries, only the tables that are explicitly defined in the configuration file will be accessible by ViDeTTe. After this configuration file is imported, (which is done directly from the notebook's UI), it will be hidden from the UI and encrypted so that this information is no longer visible to the notebook reader. Furthermore, as an added benefit, the schema of the accessible tables will be displayed on the UI, thus allowing the notebook user to get a quick glimpse on it, when generating queries.

Once this step is done, the analyst is free to issue queries in order to access database systems. Figure **??** contains a sample query that is used in our analysis. The query joins the two tables: *visitors* and *page_views* on the id of the visitor, then groups the result on the *time* attribute and runs a *count* aggregate to count the visitors per unit of time. Lastly, it sorts the resulting dataset by units of time in ascending order. Figure **??** shows the result of the query. Notice that the result has been converted into a JSON array by ViDeTTeİn the next section we will show how to use this dataset to generate our first visualization, without the use of any imperative logic.

2.1 Source Wrappers

ViDeTTe contains a set of source wrappers that enables data retrieval from various different sources both relational and non-relational. These source wrappers come pre-installed with ViDeTTe so that the notebook user will not have to take on any system administrator duties. The user simple provides the configuration file and writes the query in the language supported by each database system. The source wrapper uses that information to connect to the respective database system, retrieve the requested data, convert it into JSON format and make it available to the user. For database systems that do not contain tables with schema, the notebook user has to provide the appropriate construct under the "expose" attribute of the configuration file (i.e if the user accesses a MongoDB database, she will have to declare the collections of documents that will be consumable from the notebook). Additionally, in cases when the accessed database system does not have a schema the notebook will simply display a small portion of the dataset, thus assisting the notebook user to compose queries.

Costas: Maybe put the last two sentences in a side-note?

3. DATA CURATION

DATA CURATION

4. VISUALIZATION

4.1 Template Instance

The main challenge when generating visualizations in interactive notebooks is the variance between the APIs of the visualization libraries. For each visualization library the analyst decides to use, she has to perform an installation of the respective packages (which requires advanced technical knowledge on its own), read lengthy documentation pages that dictate how to use functions provided by each library and then engage in tedious imperative programming in order to "massage" the existing datasets into a set of formats accepted by each employed API function.

```
   sources : [ {
     driver   : "postgres",
     host     : "edu.db.domain",
     expose   : [ {
      schema : 'website_info',
      tables: [visits, page_views] } ]
     port     : 5432,
     username : "dbadmin"
     password : 'myP@ss'
   }]
```

(a) DB Access Configuration file

```
<% let readings =
   SELECT count(time) as visits, time
   FROM (SELECT * FROM page_views pv
               join visitors v
         on pv.v_id = v.vid) AS joined_table
   GROUP BY time
   ORDER BY time ASC %>

```

(b) Data retrieval

```
readings = [
   {visits: 15, time: '08:00'},
   {visits: 10, time: '09:00'},
   {visits: 25, time: '10:00'},  ...]
```

(c) Query Result

```
<% unit highcharts %>
{
  title: { text: 'Visitor information' },
  yAxis : {
    labels : [
      <% for reading in readings %>
        <% print reading.time %>
      <% end for %>],
    min : =<%= min_time
    max : =<%= max_time
  }
  series: [{
    data: [ <% for reading in readings %>
        {
          y  : <% print reading.count %>
        }
      <% end for %> ]
  }]
}
<% end unit %>
```

(d) Template `temp_view`

Figure 1: Template, template instance, and UAS configuration file for the running example

```
<% template temp_view() %>
  <% import functions %>
  <% import actions %>

  <% let readings = select t.temp
                    from db.temperature as t
                    order by timestamp %>

  <% unit highcharts %>
  {
    title: { text: 'Temperature monitor' },
    series: [{
      data: [
        <% for reading in readings %>
          {
            y    : <% print reading.temp %>,
            color: <% print toHex(reading.temp, threshold) %>
          }
        <% end for %>
      ],
      lineWidth: 1
    }],
    <% event onSelection redrawSelected() %>
  }
  <% end unit %>

  <% unit slider %>
  {
    min  : 0,
    max  : 10000,
    value: <% bind threshold = 65 %>
  }
  <% end unit %>
<% end template %>
```

(a) Template `temp_view`

```
<% unit highcharts %>
{
  title: { text: 'Temperature monitor' },
  series: [{
    data: [
        { y: 55, color: '#359435' },
        { y: 57, color: '#359823' },
        { y: 56, color: '#359533' },
        { y: 53, color: '#359220' }
    ],
    lineWidth: 1
  }]
}
<% end unit %>

<% unit slider %>
{
  min : 0,
  max : 10000,
  value: 65
}
<% end unit %>
```

(b) Template instance

```
   sources : [ {
     driver   : "postgres",
     host     : "localhost",
     port     : 5432,
     aliases  : [{db : "sensorDb"}]
     username : "dbadmin"
   }]
```

(c) UAS configuration file

Figure 2: Template, template instance, and UAS configuration file for the running example

ViDeTTe Visual Units

To shield analysts from this laborious task, ViDeTTe abstracts out each visual component as a ViDeTTe *visual unit* (or simply *unit*). In the eyes of an application developer a visual unit is simply a black box that takes as input a JSON value containing the data that describe the visualization and simply constructs the respective visual component. As such a particular instantiation of the unit (denoted as *visual unit instance* or simply *unit instance*) can be described as **<% unit** U **%>** v **<% end unit %>**, where U the type of the visual unit and v the JSON value corresponding to the input of the unit.

==== Edited up to here =====

For instance, lines 1-14 of Figure **??** show a unit instance of type `highcharts`. The unit instance specifies both the data that should be shown on the chart (as an array of tuples assigned to the `data` attribute) and the values of the parameters accepted by the unit (such as the chart's title).

As we will discuss in Section **??**, visual units are generated by visual unit developers who wrap popular visualization libraries (such as Google Maps, HighCharts, etc) and make them available to application developers through ViDeTTe's visual unit library. In addition to the units provided by unit developers, ViDeTTe also provides a special unit of type *HTML*. In contrast to other units, which expect JSON values, an HTML unit takes as input a sequence of HTML tags, allowing developers to easily create simple HTML content. For ease of exposition we focus in this paper on JSON units. Our results can be easily extended to the HTML unit. *Costas: So far we've been talking about JSON++ if we keep that, we should make it clear why in this paragraph we only talk about JSON and HTML* *YannisK: Removed HTML unit encompassing the two JSON units from our running example*

Figure **??** shows the grammar for template instances. Finally, to simplify the architecture, template instances are internally represented as JSON++ values. This is achieved by encoding a list of unit instances **<% unit** U_1 **%>** v_1 **<% end unit %>** ... **<% unit** U_n **%>** v_n **<% end unit %>** as a JSON++ tuple {U'_1: v_1, ..., U'_n: v_n}, where U'_i is a unique identifier assigned by ViDeTTe to the i-th unit instance. *YannisK: OK or too cryptic?*

YannisP: We need reserved keywords for the attribute names that correspond to units. Eg, _myunit *Costas: How can we have a predifined set of reserved keywords given that there's an arbitrary number of possible units each with a unique name?*

4.2 Template

YannisP: About Figure **??***: the syntax allows a single top level unit, which is probably correct but not what the example does. Is the top level unit always html?*

YannisK: To do: Modify template BNF grammar to make sure that we can have directives appear inside JSON values *Templates* are declarative specifications describing the template instances as a function of UAS data. A template specifies this function through a set of *template directives*, so that it provides syntax similar to well-known template languages, while it is essentially an expression of a functional programming language without recursion. A template also specifies collecting data from the page and catching events *Costas: perhaps we mean linking event with corresponding actions?* .

There are five template directives: **<% print %>**, **<% for %>**, **<% let %>**, **<% bind %>**, and **<% event %>**. These are

```
1  template   →  <% template template_name (param_list) %>
                   let*
                   unit
                 <% end template %>
2  param_list →  ( var_name (, var_name)* )?
3  unit       →  <% unit unit_class %>
                   value
                 <% end unit %>
4  value      →  jsonpp_value
5             |  unit
6             |  print
7             |  [ for ]
8             |  < for >
9             |  if
10            |  bind
11            |  { event*
                   ("string" : value
                   (, "string" : value)* )? }
12 let        →  <% let var_name = expr %>
13 print      →  <% print expr %>
14 for        →  <% for var_name in expr %>
                   let*
                   value
                 <% end for %>
14 if         →  <% if expr %>
                   value
                   (<% elif expr %>
                   value)*
                   (<% else %>
                   value)?
                 <% end if %>
15 bind       →  <% bind var_name = expr %>
16 event      →  <% event event_name action_name %>
17 expr       →  js_expression
18            |  source_expression
19            |  json_path
```

Figure 3: BNF Grammar for Templates

used to describe computation, define variables, set up data collection and specify events. We next describe each of them in detail.

YannisP: From an expressiveness point of view, let does not add anything. The interesting consideration is whether it has a meaning about IVM: materialized vs virtual intermediate result.

Defining variables. A template may define variables that are added to the UAS instance so that they can be used in subsequent computation. Variable definition is facilitated by the **let** directive.

The **<% let** x **=** E **%>** directive defines variable x in the UAS, and assigns to x the result of evaluating the expression E. The expression E can be a JavaScript expression, a source-specific language (such as a SQL query in the case of relational database sources) or a JSON++ path. For example, the template of our running example (lines 6-8) employs a **let** directive to create a variable `readings` containing all temperature readings (retrieved from a relational DBMS through a SQL query).

Reporting syntax and semantics. Computation in a template is specified using the **print** and **for** directives. The **<% for** x **in** E **%>** B **<% end for %>** directive specifies that variable x iterates over the result of the expression E. In each iteration, the body B of the **for** loop is instantiated. For example, the template of our running example (lines 14-19) uses a **for** directive to iterate over the sensor readings (stored in the `readings` variable). For each reading, it generates a new JSON tuple of the form `{y:..., color:...}`, which is the data format expected by the HighCharts unit for each data point.

The **<% print** E **%>** directive instantiates the result of expression E. For example, the template of the running example uses two **print** directives to generate the values of the `y` and `color` attributes of each JSON tuple produced by the **for** directive. *YannisP: The part between the "{}" could be removed* {The value of `y` is created by printing the value of the `reading` variable (line 16), while the value of `color` is generated by calling the `toCSS` function, which takes as input the current reading and the normal temperature (set through the slider) and produces a CSS color code according to the coloring schema explained above (line 17).}

Collecting data. In addition to specifying how to compute a template instance, the template's **bind** directive allows the developer to specify how data are collected from user input on the page.

The **<% bind** $x = E$ **%>** directive specifies that the template instance attribute value in whose position the directive appears will be assigned to variable x. This allows UAS variables to become bound to input received by visual units. *YannisP: If UAS has not been defined we need to say what is the JavaScript target.* *Costas: We should define a Model/VDB variable, that encompasses every client-side variable used in an application. The target in this case would be the Model.x variable* For instance, the template of our running example (line 30) uses a **bind** directive to assign to variable `threshold` the current value of the slider (which is returned by the slider unit as the value of the `value` variable). The **bind** directive also allows the developer to specify an expression E, whose value will be assigned to the variable when the template is first instantiated. For example, the slider of our running example is initialized with the value 65.

Specifying event handlers. Finally, a template allows also the invocation of actions in response to events. The **<% event** e a **%>** directive specifies that whenever event e of the enclosing unit instance occurs, action a is executed. For instance, *YannisK: Costa, please add event to template and add description here.*

Costas:

Finally, each template may also accept parameters that are passed to it by its caller by value. When the template is instantiated, ViDeTTe creates for each parameter an identical-named UAS variable and then sequentially scans the template instantiating its directives.

5. DISCUSSION

DISCUSSION

6. CONCLUSION

7. REFERENCES