GSoC 2016 Application
Organization: TARDIS-SN
Proposal Title: Carsus - TARDIS support package for creating atomic datasets.

Mikhail Mishin

March 2016

1 Basic Information

Contact Information:

– **Name**: Mikhail Mishin

– **Email**: mishinma1805@gmail.com

– **University**: Bauman Moscow Technical State University (BMSTU)

– **Phone Number**: +79154089955

– **Github/ Gitter Chat Room**: mishinma

– **Skype**: mishinma1805

Relevant Skills:

– Two years' programming experience in python

– Experience in relational database design

– Familiarity with popular python packages, including pyparsing and sqlalchemy

– Good background in physics, especially quantum physics

2 Project Proposal Information

2.1 Introduction

For running its simulations TARDIS needs atomic data, such as atomic masses, ionization energies, levels and transitions. This data is available from a number of known sources. In order for TARDIS to be able to use the data, it should be downloaded, stored in a database, and converted to the HDF5 format. This task is currently done by the `tardisatomic` package; however, this package has some underlying issues. It has no clear workflow and the existing database structure is difficult to modify and maintain. These problems complicate adding and processing data from new sources.

If my project is accepted, I will implement a new package named `carsus` (the name was suggested by the collaboration) that will be deprived of the issues inherent in `tardisatomic`. The new package will have clear workflow, sound database structure and flexible input/output modules.

2.2 Mission Statement and Objectives

Mission statement: The purpose of the new `carsus` package is to prepare atomic datasets for TARDIS and other organizations.

Mission objectives:

1. Download data from the sources specified in the TEP 004.
2. Store that data in a common format within the relational database. The database main objectives are:
   a. Maintain all ingested atomic data
   b. Allow ingesting data related to the same object from different sources.
   c. Keep track of all used data sources and units.
   d. Allow for interactive exploration
   e. Produce the data requested by the output scripts
3. Read the database and output the data in the HDF5 format or any other format
4. Adding new parsers and data sources to the package should be straightforward

2.3 Implementation

I have already started working on `carsus` and it is available at my repository `https://github.com/mishinma/carsus`. `carsus`, as well as TARDIS, is an Astropy affiliated package, so their layouts are similar. The source code is located in the directory with the same name as the package's name - `carsus`.

2.3.1 Input modules

As suggested in the TEP004 [1] input and output modules should be located within `carsus/io`. There should be a separate package (directory) for each of the sources, e.g. parsers for the NIST databases would be in the `carsus/io/nist` package.

Three kinds of objects are used in the input modules: download functions, parsers and ingesters:

• A download function downloads data from the source.

• A parser parses this data into a pandas DataFrame. In many cases additional processing of this DataFrame is needed before ingestion, e.g. the extraction of nominal values and standard deviations from strings like "1.02342[12]". Also a DataFrame can contain data related to different subjects from the database, e.g. atoms and isotopes. Such a DataFrame should be broken into smaller DataFrames that contain data related to only one subject. So a parser has tasks other than simple parsing, and this sets the rationale for using the BaseParser class. Each data source has its own Parser class that inherits from the BaseParser class. Figure 1 illustrates the inheritance diagram.

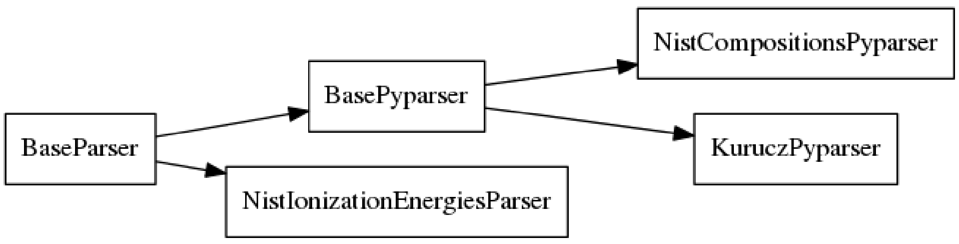

Figure 1:

Pseudocode for the BaseParser class:

```
class BaseParser(object):
    def __init__(self, columns)
        self.columns = columns
        self.base_df = pd.DataFrame()
    def load(self, input_data):
        # parse the data and store the results in the base_df
        pass
```

A parser that inherits from the BaseParser parses data into a DataFrame and stores it as its `base_df` attribute. A parser can use the `base_df` to create other DataFrames tailored for ingestion. For example, if the `base_df` contains data related to *isotopes* and data related to *atoms*, an *isotopic* DataFrame and an *atomic* DataFrame could be prepared. Initially the `base_df` is empty; one need to *load* parser with data.

There are several options for parsing data; in each case the most appropriate method should be identified and used. Pandas input functions can deal with preformatted tables and csv files, beautifulsoup can parse html pages and pyparsing can be used in non-trivial cases. When using pyparsing I will use the best practices from the Getting Started with Pyparsing book [3] such as: defining the grammar in the BNF notation before implementing it; setting parse actions to process the input at parsing time and setting tokens' names. Parsers that use pyparsing inherit from the BasePyparser class. It defines additional attributes such as grammar and also it defines the load method. I have already implemented the input module for the NIST Atomic Weights and Isotopic Compositions Database using pyparsing. Refer to my commit note for the details.

•An ingester persists data to the database. As with parsers, each data source has its own Ingester class that inherits from the BaseParser class. Figure 2 shows the inheritance diagram:

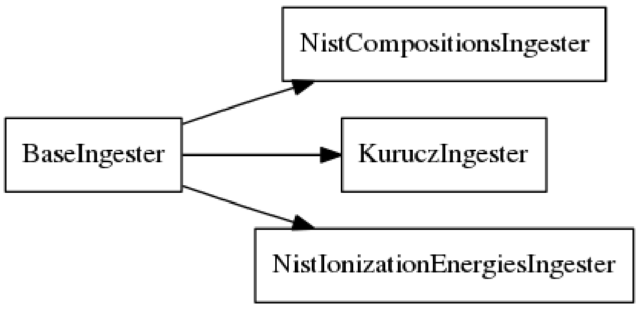

Figure 2:

Pseudocode for the BaseIngester class:

```
class BaseIngester(object):

def __init__(self, session, parser, downloader):
    self.session = session
    self.parser = parser
    self.downloader = downloader

def download(self):
    pass

def ingest(self):
    pass
```

An Ingester instance establishes conversations with the database using its session attribute (the session is configured in another module). It also has a parser and downloader that it uses to get the data. The typycal ingest workflow is as follows:

1. User creates an ingester instance by calling the data source's Ingerster class with the configuration parameters, e.g: `ingester = NistAtomicWeightsIngester(exclude_atoms=list_of_excluded_atoms)`

2. User calls the ingest method: `ingester.ingest()`. Then, the ingester instance:

   a. Downloads the data with its downloader function.
   b. Loads its parser with data.
   c. Determines what is going to be ingested (e.g isotopic data)
   d. Asks its parser to prepare the corresponding DataFrames (e.g. an isotopic DataFrame).
   e. Ingests data from the returned DataFrames.

Another important class is the AtomicDatabase class. It establishes the connection to the database, creating the session binded to the database. The session object is stored as an attribute of AtomicDatabase instance. In this way the *session scope* begins when an AtomicDatabase instance is created and ends when it is garbage-collected. The *transition scope* begins when the ingest method is called and ends after it is completed.

2.3.2 The database

`tardisatomic` used to use the Python DBAPI specification to access the database, but this proved to be not scalable. I have analyzed the workflow of this approach and its issues in the New TARDIS atomic database document. The second approach used was to access the database using the SQLAlchemy Object Relational Mapper (this is also suggested in the TEP004 [1]. SQLAlchemy provides a "consistent and fully featured facade over the Python DBAPI" [4] that liberates us from having to insert SQL statements into the code. This improves readability and testability of the code; besides, if we wanted to migrate the database to another SQL dialect, the changes to the code would be minimal. The SQLAlchemy ORM associates python classes with database tables, simplifying the development of the database and making interactions with it easy and "pythonic". Taking everything into account I support the team's choice to use SQLAlchemy in Carsus. The code related to SQLAlchemy resides in the carsus/alchemy package.

Before implementing the database I plan to conduct a full database design process following these steps:

1. Define statement and objectives. These are defined as a part of the package objectives in 2.2.
2. Analyze the current database and data sources. I have already examined the legacy database structures from `tardisatomic` and written some conclusions in the New TARDIS atomic database.
3. Define the preliminary table structures and establish field specifications.
4. Determine the relationships between tables and establish each relationship using primary keys and foreign keys.
5. Determine the constrains that must be imposed upon the data and establish them.
6. Review the final database structure for data integrity.

The steps are taken from the well-known book *Database Design for Mere Mortals* [2]. I will conduct the database design process concurrently with implementing the parsers for the main data sources. Once the database schema for the main data sources is created, it can be modified to store information from other sources.

As a preliminary task I have implemented the parser for the NIST Atomic Weights and Isotopic Compositions database. The code is currently on the nist_comp branch. To store the parsed data I created a simple database schema shown in Figure 3.

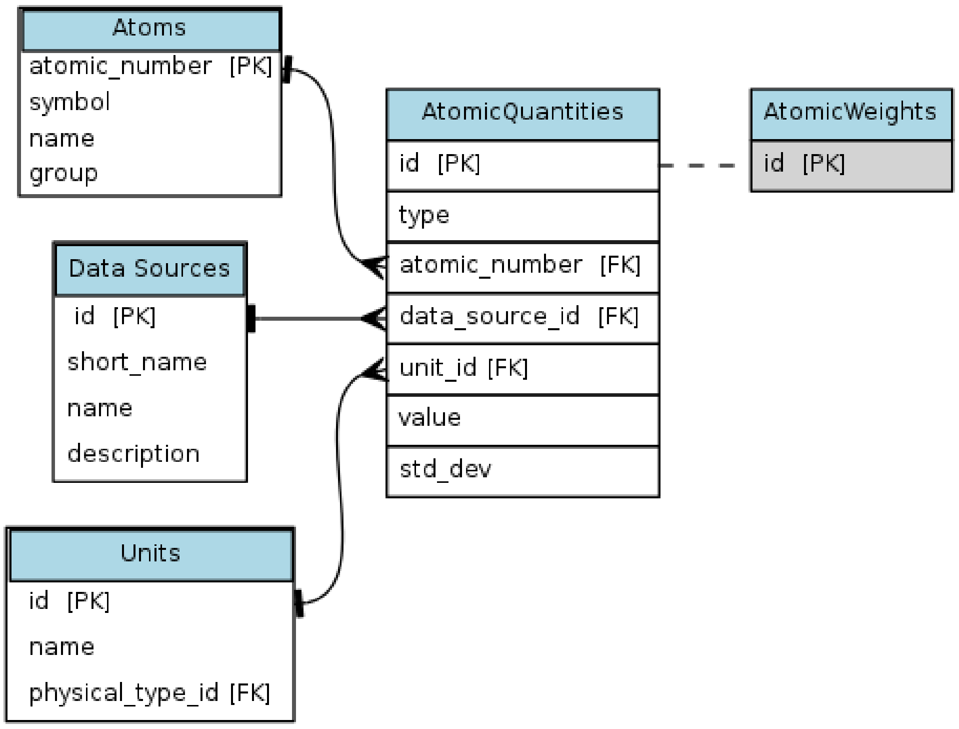

Figure 3:

The scheme is trivial except for the `AtomicQuantities` table. This table establishes the one-to-many relationship between an atom and its quantities. The `AtomicWeights` table is a subset table of the `AtomicQuantities`. It represents the special type of quantities related to atoms - atomic weights. The `AtomicWeights` table contains all fields germane to the `AtomicQuantities` table and even shares its primary key. Using this schema we can store atomic weights that are related to one atom and that are from different sources. There is a unique constraint on `atomic_number` and `data_source_id` from the `AtomicWeights` table; so there cannot be two quantities of the same type that are related to the same atom and are from the same data source. This schema preserves data integrity because there is no duplication and all tables are normalized. SQLAlchemy supports inheritance and that allows us to implement subset tables. You can see my implementation of the schema in the carsus/alchemy/atomic.py module. I used the single table inhreritance configuration, where "the attributes of the base class as well as all subclasses are represented within a single table".

2.3.3 Output modules

I plan to brainstorm the implementation of output modules after the database has some data in it. One my of my ideas is to define for each SQLAlchemy class a corresponding class that can query the database and output the data in needed formats (e.g. HDF5).

2.3.4 Configuration options

I plan to add configuration options as command-line parameters using the standard argparse module. Later the support of configuration files can be added.

2.3.5 Documentation and testing

The documentation for Carsus will be created using sphinx and will follow numpy documentation guidelines. The testing will be done with pytest employing parameterized tests, fixtures and mock testing.

2.4 Schedule

Total duration of coding period: 12 weeks: May 23 - August 23. The primary goal is to make `carsus` work with all `tardisatomic` data sources while meeting all objectives defined 2.2. The secondary goals are to add new data sources, add functionality to the output module and more configuration parameters. Tests and documentation are already taken into account in the schedule.

Bonding period: April 22 - May 22

- familiarize myself with relevant topics from astronomy and physics
- discuss in detail with the team the proposed structure of the input modules
- go through steps 1-3 from the section 2.3.2. Discuss each step with the team.
- plan how to implement parsers while examining the data sources in the previous step. For each source decide whether to use pyparsing, pandas input methods or beautifulsoup. Define the BNF grammar if pyparsing is chosen.
- at the beginning of the coding perioud I should have the preliminary database schema and know how to implement the parsers for main data sources.

Coding period:

- Week 1: May 23 - May 29: finish the work on nist_comp branch and merge it. Add configuration options.
- Week 2: May 30 - June 5: implement the parser and mapping SQLAlchemy classes for Kurucz
- Week 3: June 6 - June 12: finish the work on the Kurucz data source. Define and implement configuration options.
- Week 4: June 13 - June 19: implement the parser, mapping SQLAlchemy classes, and configuration options for NIST ionization energies.
- Week 5: June 20 - June 26: buffer week, polish the parsers, review the database structure, finish tests and documentation. Start the discussion about the best way to implement output modules.

- Week 6: June 27 - July 3: implement the chianti data source in the same manner.
- Week 7: July 4 - July 10: finish the implementation of the chianty data source. Define in detail the design of output modules and output configuration options, e.g to exclude atomic quantities from one source and include them from the other source.
- Weeks 8-11: July 18 - August 24: implement the output modules. If there is time add optional data sources.
- Week 12: August 15 - August 23: buffer week, prepare for the final evaluation.

Optional data sources: Norad and CMFGEN.

3 About Me

I am a fourth year bachelor student from the Computer Science faculty of Bauman Moscow Technical State University. I have been using python for two years, mostly for implementing scientific scripts and writing small packages. What I really like about the language is that it has its own philosophy, that can be a great guidance sometimes.

I am also very fond of science, especially physics, and that is why I chose TARDIS as my GSoC organization. The atomic datasets project has really engaged me and I am intended to work hard to complete it. I do not have any other commitments during the summer except for my graduation in the first week of June. During this week I may not be able to work full time, but otherwise I plan to work at least 50 hrs/week. During the bonding period I will be studying at the university and I will be able to work maybe 30-35 hrs/week. I also have some exams in the beginning of May and may be inactive during this period.

4 Links to patches and contributions

1. The nist_comp branch of the Carsus package `https://github.com/mishinma/carsus/tree/nist_comp`; see the commit note
2. Pull request to the cassis package `https://github.com/tardis-sn/cassis/pull/15` see the commit note
3. The New Tardis Atomic Database document `https://www.overleaf.com/read/wrcsvvwxbdtj`
4. Opened this issue `https://github.com/tardis-sn/tardis/issues/478` and closed it with this PR `https://github.com/tardis-sn/tardis/pull/479`
5. Opened this issue `https://github.com/tardis-sn/tardis/issues/483`
6. Made this PR for the testing project `https://github.com/tardis-sn/tardis/pull/482`

References

[1] TARDIS collaboration. *TEP004: TARDIS Atomic improvement*, 2016 `https://github.com/tardis-sn/tep/blob/master/TEP004_tardisatomic_restructure.rst`

[2] Hernandez, Michael J. *Database Design for Mere Mortals: A Hands-On Guide to Relational Database Design (3rd Edition)*. Addison-Wesley, 2013.

[3] McGuire, Paul. *Getting Started with Pyparsing*. O'Reilly Media, 2007.

[4] Bayer, Mike. *Introduction To SQLAlchemy And ORMs*. Pycon US 2013. Presentation.