Table of Contents

Please Note: This is a ***pre-release*** *of the documentation for Simple VIVO and the Pump. The documentation is being written as the software is developed. Some elements of the software are incomplete, or have not been thoroughly tested. Some elements of the documentation are not complete, have not been reviewed and have not been copy-edited. The book is changing by the day, so please stop back to see how it has been improved. M. Conlon*

Simple VIVO

Simple VIVO is a command line tool for managing data in VIVO using spreadsheets. With Simple VIVO you can get the data from your VIVO into spreadsheets on topics such as grants, people, publications, educational training, and more. You can then edit those spreadsheets and update VIVO with your edits.

Simple VIVO is open source and uses a software tool called the Pump to get data from and update data in VIVO.
The Pump is included with Simple VIVO, and can be used by programmers to create new data management tools.

You can download Simple VIVO and the Pump, along with examples and sample data from GitHub

New to VIVO? VIVO is an open source, community developed, semantic web application for managing an integrated view of the scholarly work of an organization. VIVO is a membership supported project of Duraspace.
You can download VIVO from GitHub

Preface

VIVO is a semantic web system for creating and maintaining an integrated view of the scholarly work of your organization.

As a semantic web application, VIVO uses an ontology to represent concepts. It uses triples to express assertions about scholarship. It uses RDF to represent those triples. The triples constitute a graph of the data regarding your organization's scholarship. Finally, it uses a triple store to store the triples that constitute the graph. All of these technologies (VIVO, ontology, triples, RDF, graph, and triple store) may be new to you and your organization.

Since 2009, VIVO has become available more broadly, and is now in use at more than 100 organizations world-wide. The technologies it uses are still new to many, and can be very challenging. Many data managers and traditional Information Technology departments have struggled to master the technologies, particularly those for entering and updating data in bulk. Large organizations may have the manpower to commit to learning about the semantic web, building tools, deciphering VIVO's information representation models and eventually provide capabilities for bulk loading and managing their data.

Many organizations have developed custom solutions for what has become known as "the ingest" problem -- mapping traditional relational data sources to VIVO's graph models, and providing the means for adding, updating and removing data from VIVO based on authoritative sources of information such as the organization's human resource data, and the registrar's records of teaching.

Two existing solutions for the ingest problem should be noted. 1) The VIVO Harvester is a software tool that has been used by IT departments to perform VIVO data management from authoritative sources. It requires professional IT staff and deep knowledge of the VIVO models. 2) Karma is an open-source, web-based tool that can be used to develop maps that allow data to be managed in web-based rectangles and uploaded to VIVO. Karma can be used by data managers who must master the VIVO data models to build Karma maps.

Despite its complexity and the dearth of data management tools, VIVO has prospered because it is open, and because its data model is extensible. This extension capability is a blessing and a curse. Extending the data model allows users of VIVO to represent any form of scholarship using any constructs they desire. The downside is that VIVO sites may diverge and become unable to share data, and standard tools for data management can not anticipate the diversity of extensions.

Simple VIVO and is provided to fill a gap in VIVO data management practice. Simple VIVO allows data managers to manage data from spreadsheets or any text editor capable of producing tab-delimited data, using standard VIVO data models. Simple VIVO does not require mastering VIVO data models nor does it use any VIVO extensions. Examples are provided for all the common elements of VIVO data management. Using Simple VIVO you can create a VIVO from scratch, fill it with data about your scholarship, manage the data as it changes, and enjoy the benefits of VIVO -- open access to data regarding your scholarship, integrated data at the individual level for analysis, the ability to easily share your data with others, and presence in a growing community of scholars, data managers, and technologies working to create next generation scholarly ecosystems.

Managing data in VIVO was too difficult. Only the most determined "early adopters" would commit to learning the underlying technologies, and the data models. And only these early adopters would further develop their own custom methods for managing data in VIVO.

With Simple VIVO, that has changed. Simple VIVO provides the means for any group to manage VIVO data. If you can use a spreadsheet and you understand basic concepts of data management, this book, and Simple VIVO, are for you.

We are trying to keep things simple. If you find that some explanation is not clear, or some process is complex, please share your experience.

Two sets of examples are provided with Simple VIVO.

The first set of examples, the "generic" examples, show how to use Simple VIVO to manage all elements of a VIVO profile in a scholarly organization without resort to additional software or VIVO extensions. These examples should provide you with the means to manage your data in VIVO.

The second set of examples are from the University of Florida. The University of Florida is one of the largest universities in the United States with more than 6,000 faculty, 50,000 students, and more than 950 academic and administrative departments. Each year, the university teaches more than 12,000 courses, receives more than 3,000 grants, publishes more than 6,000 papers, and has more than 2,000 new employees. Simple VIVO was designed to manage the data of the University of Florida. University offices produce data on teaching, grants, and employment. The examples provided with Simple VIVO show how data received from these offices can be managed in VIVO using Simple VIVO. These examples use University of Florida ontology extensions.

I hope you find Simple VIVO useful. And I hope to hear from you about your use of Simple VIVO.

Mike Conlon, Gainesville, Florida

How to Read This Book

Who has time to read a book? We want to read what we need and get going. This book can be read from cover to cover, but that's not necessary. Here's what you need to read:

Data Managers

Read Simple VIVO and the chapters pertaining to the materials you want to manage. Not every VIVO site will manage all the materials.

Programmers

Read the Preface and skip to The Pump and all that follows.

System Administrators

Read Getting Started. For your data managers and programmers to use Simple VIVO, you will need to enable the VIVO SPARQL API. See the VIVO documentation. You will need to provide the API URI, and a username and password for an account authorized to read and write VIVO data through the API.

You may also need to help your data manager with a command line environment suitable for running Python programs such as Simple VIVO.

Getting Started with Simple VIVO

To start using Simple VIVO to manage data in your VIVO, you will need the following:

1. A VIVO to use for data management. VIVO is typically deployed at the organizational level, much like a web server. Simple VIVO is used to manage the data in an existing VIVO. The existing VIVO may be "empty" (as delivered VIVO, has a small amount of data pertaining to countries, states and territories of the United States, and data about its own definitions). Simple VIVO can also be used with VIVO Vagrant, a VIVO which can be installed on your local machine. Your VIVO must allow the VIVO SPARQL API to be used. By default, this is turned off. Your system administrator will need to turn it on and provide you with a username and password for you to use to manage data in your VIVO.
2. You will need to be able to run Python 2.7 from a command line. If this is new to you, a system administrator can help you prepare your machine, and show you how to use the command line.
3. You will need to have the VIVO Pump and Simple VIVO installed on your machine. This is very simple to do. Once Python is installed, you can type: `pip install vivo-pump` at the command line prompt.
4. You will need to edit the Simple VIVO configuration file `sv.cfg` to provide four pieces of information:
   i. The URL of your VIVO SPARQL API. Your system administrator will have this value
   ii. The username required to access your VIVO SPARQL API
   iii. The password required to access your VIVO SPARQL API
   iv. The URI pattern you would like Simple VIVO to use when created new URI (see below)

With these things in place, you are ready to manage data in VIVO using spreadsheets.

URI patterns

In VIVO, everything has a URI, a Uniform Resource Identifier. Simple VIVO can be used to create new things in
your VIVO, adding grants, or people, or dates, or educational background. Everything that is added will have its
own URI. Simple VIVO makes URI that look like:

```
PatternNumber
```

Where pattern is a pattern you supply, and Number is a random number generated by Simple VIVO for each new thing
it creates. Simple VIVO checks to make sure the number it is create
One is not already in use in your VIVO.

Your pattern should like `http://yourvivourl/individual/n`

Let's looks at each part of the pattern and see what options might exist for you

`http://` This is required. All VIVO URI are addresses on the semantic web.

`yourvivourl` Many organizations have VIVO URLs that look like `VIVO.school.edu` or something similar. Your
system administrator can provide the URL of your VIVO.

`individual` This is the VIVO default. Some VIVO sites use alternate words. Your system administrator can
tell you what word is used at your site.

`n` This is the default. You can use any word or letter you like. You may prefer that dates have uri with
the word date. Person might be used for uri for people.

Let's put all the pieces go ether in an example.

Example: The University of Florida has a VIVO at `http://VIVO.ufl.edu`. They use the word individual. They
also have all their VIVO uri starting with n. Their uri pattern for all their Simple VIVO work is

```
http://VIVO.ufl.edu/individual/n
```

Simple VIVO will make uri that look like `http://VIVO.ufl.edu/individual/n883882`

Example 2: mythical university has a VIVO at `http://profiles.mythical.org`. They use the word "u" to refer to
all uri in their system. They use different prefixes for uri referring to different things.

```
http://profiles.mythical.org/u/date is the prefix they use for dates
http://profiles.mythical.org/u/grant is the prefix they use for grants
```

And so on

Each uri prefix is put in the config used to manage each type of thing.

Simple VIVO will make uri. That look like `http://profiles.mythical.org/u/date774883` for a date,
and `http://profiles.mythical.org/u/grant884001` for a grant.

Testing your set up

Once you have all the pieces in place -- a VIVO, Python, the Pump and Simple VIVO, and the four parameters in
your `sv.cfg` file, you can test your setup by going to a command line, and typing:

```
python sv.py -a test
```

`sv.py` is the name of the Simple VIVO program.

The word `python` is required. It tells the system "I'm asking you to run a Python program and here's its name"

The `sv.py` is required. It tells python "the name of the program I would like to run is `sv.py` and you can
find it in the current directory." If `sv.py` is not in the current directory, you can change to the directory it is
in to run it, or use a path to indicate where `sv.py` is located.

The `-a test` command line parameter tells Simple VIVO that you would like to run the test to see that Simple
VIVO is set up correctly.

If you have python installed and `sv.py` where python will find it, and you have typed the command line correctly,
Simple VIVO will respond with a list of the parameters configured for Simple VIVO and the results of a connection
test with VIVO. For example:

```
mymac$ python sv.py -a test
2015-10-28 10:26:31.231997 Start
2015-10-28 10:26:31.319732 Test results
Update definition    pump_def.json read.
Source file name    pump_data.txt.
Enumerations read.
Filters    True
Verbose    False
Intra field separator    ;
Inter field separator    |
VIVO SPARQL API URI    http://localhost:8080/vivo/api/sparqlQuery
VIVO SPARQL API username    vivo_root@school.edu
VIVO SPARQL API password    xxxxxxxx
VIVO SPARQL API prefix
PREFIX rdf:      <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs:     <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd:      <http://www.w3.org/2001/XMLSchema#>
PREFIX owl:      <http://www.w3.org/2002/07/owl#>
PREFIX swrl:     <http://www.w3.org/2003/11/swrl#>
PREFIX swrlb:    <http://www.w3.org/2003/11/swrlb#>
PREFIX vitro:    <http://vitro.mannlib.cornell.edu/ns/vitro/0.7#>
PREFIX wgs84:    <http://www.w3.org/2003/01/geo/wgs84_pos#>
PREFIX bibo:     <http://purl.org/ontology/bibo/>
PREFIX c4o:      <http://purl.org/spar/c4o/>
PREFIX cito:     <http://purl.org/spar/cito/>
PREFIX event:    <http://purl.org/NET/c4dm/event.owl#>
PREFIX fabio:    <http://purl.org/spar/fabio/>
PREFIX foaf:     <http://xmlns.com/foaf/0.1/>
PREFIX geo:      <http://aims.fao.org/aos/geopolitical.owl#>
PREFIX obo:      <http://purl.obolibrary.org/obo/>
PREFIX ocrer:    <http://purl.org/net/OCRe/research.owl#>
PREFIX ocresd:   <http://purl.org/net/OCRe/study_design.owl#>
PREFIX skos:     <http://www.w3.org/2004/02/skos/core#>
PREFIX vcard:    <http://www.w3.org/2006/vcard/ns#>
PREFIX vitro-public: <http://vitro.mannlib.cornell.edu/ns/vitro/public#>
PREFIX vivo:     <http://vivoweb.org/ontology/core#>
PREFIX scires:   <http://vivoweb.org/ontology/scientific-research#>
Prefix for RDF file names    pump
Uriprefix for new uri    http://vivo.school.edu/individual/n
Sample new uri    http://vivo.school.edu/individual/n2898507243
Simple VIVO is ready for use.
2015-10-28 10:26:31.949635 Test end
2015-10-28 10:26:31.949723 Finish
mymac$
```

Notice toward the bottom "Simple VIVO is ready for use." You're good to go!

If things aren't quite right, the test will respond with one or more error messages. The most common errors
have to do with specification of

the configuration file for accessing your VIVO SPARQL API. In some cases, your VIVO may not be accepting
connections, or your username and password may not be authorized, or your uri pattern may be invalid.
Work with your system administrator to get VIVO set up properly and the parameters needed to access VIVO set up
properly in your `sv.cfg` file.

Tests passed and good to go

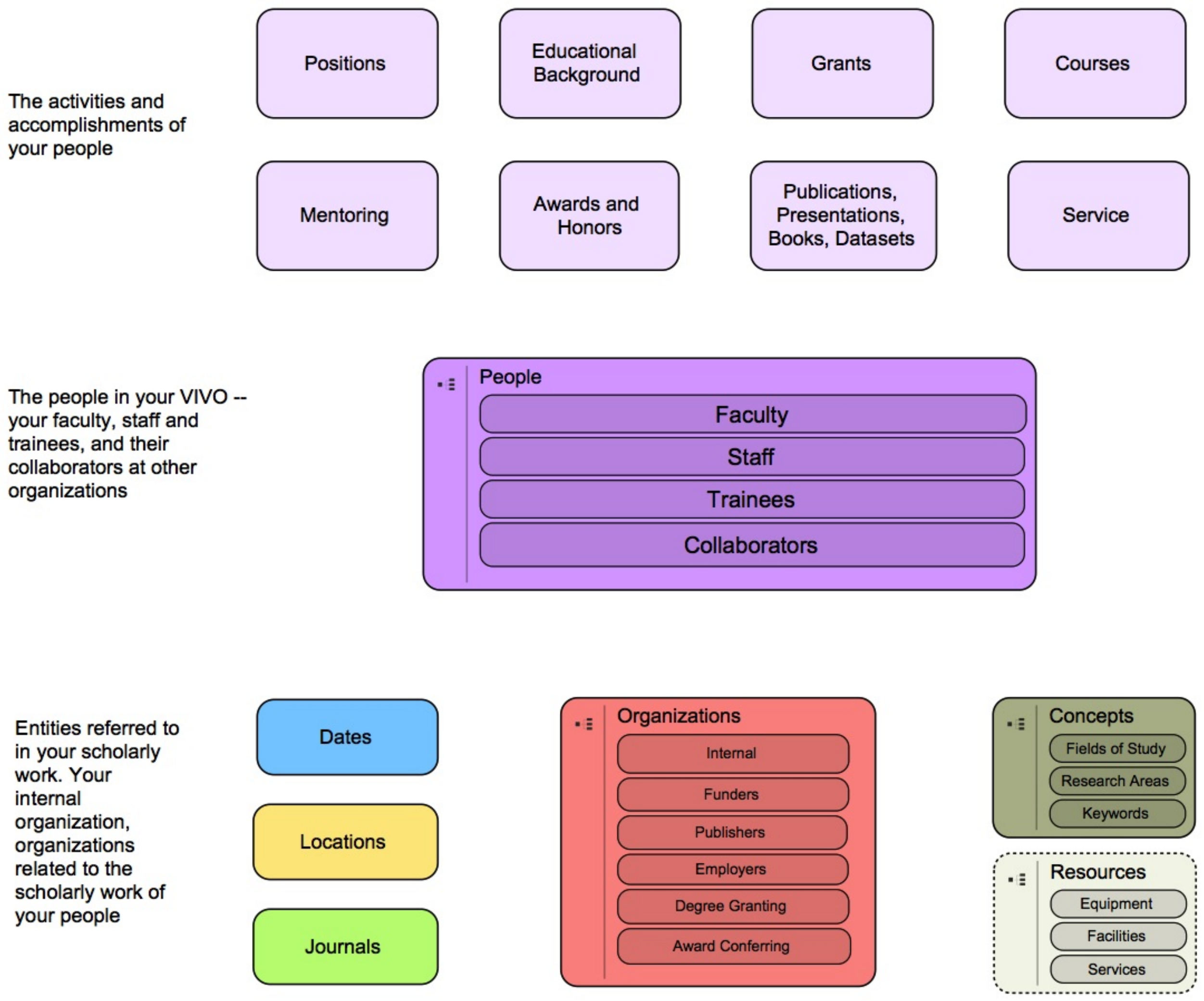
With tests passed, you are ready to use Simple VIVO. Please read the introduction to Simple VIVO that follows, and then try one or more examples.

Simple VIVO

VIVO provides an integrated view of the scholarly work of your organization. The scholarly work of your organization is complex -- faculty, research staff, their activities and accomplishments. And these are connected to each other and to institutions, journals, dates, and concepts.

Let's think about VIVO as being about people. In the figure below, we see people in the center of the diagram. Things "below" the people are things that exist in the academic environment and can exist in your VIVO without reference to people. As we think about managing data in VIVO, and in building a VIVO, these things represent the place to start. We can put these things in VIVO and expect them to be there when we begin to put in people and journals.

The things at the top of the figure are the details of the scholarly record of people. They represent the things that will appear on a person's curriculum vitae.


The activities and accomplishments of your people
Positions
Educational Background
Grants
Courses
Mentoring
Awards and Honors
Publications, Presentations, Books, Datasets
Service
The people in your VIVO -- your faculty, staff and trainees, and their collaborators at other organizations
People
Faculty
Staff
Trainees
Collaborators
Entities referred to in your scholarly work. Your internal organization, organizations related to the scholarly work of your people
Dates
Locations
Journals
Organizations
Internal
Funders
Publishers
Employers
Degree Granting
Award Conferring
Concepts
Fields of Study
Research Areas
Keywords
Resources
Equipment
Facilities
Services

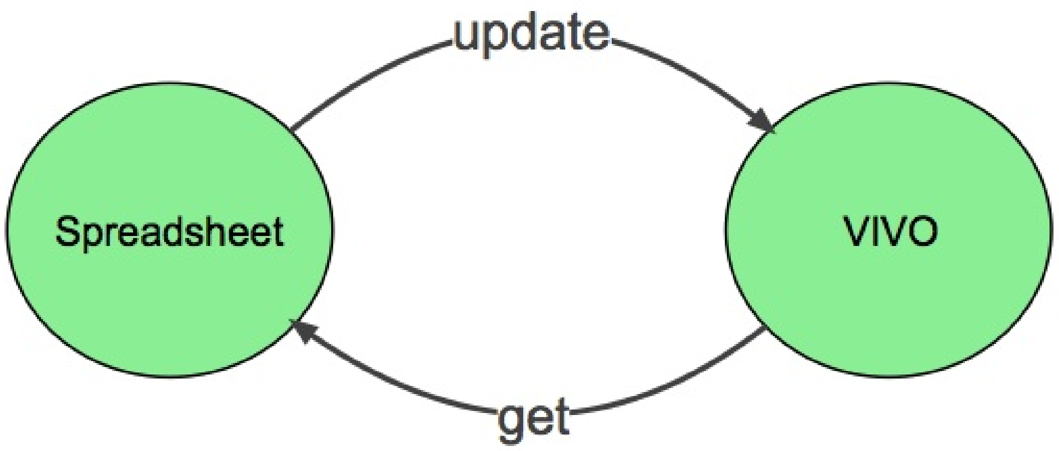
VIVO is great at representing all these things and more. But to keep it "simple," we will focus on the domains in the diagram.

The Basic Idea

What if we could manage the data in VIVO using spreadsheets?

That's the basic idea behind "Simple VIVO" a tool for using the VIVO Pump to take spreadsheet data and put into VIVO (called an update) and to get data from VIVO and put it in a spreadsheet (called a get).

The cycle of data management is shown below. Using Simple VIVO, we can "get" domain data from VIVO into a
spreadsheet. The spreadsheet data will reflect one of the domains -- people, mentoring, teaching, etc. Once the data is in a spreadsheet, it is very easy to edit. You can scan the columns looking for missing or incorrect data. You can edit the data in the spreadsheet, adding missing values, updating values, and adding rows to represent new things. You can then use Simple VIVO to update the data in VIVO using the
data in the spreadsheet as authoritative.



Spreadsheets have rows and columns. The rows in the spreadsheet will correspond to "things" in VIVO. Depending on the scenario, your spreadsheet might contain people, publications, grants, courses or other kinds of things.

The columns in your spreadsheet will correspond to attributes of things in VIVO. So, for example, if you are working with people, your columns might contain attributes such as "name" and "phone number".

For each row in each column, your spreadsheet has a cell. Cells correspond to the values that will be in VIVO. In Simple VIVO you can have one of three different things in each cell:

1. Your cell contains a value. That value will be used to update VIVO.
2. Your cell contains the word "None." None is a special word in Simple VIVO. It means that VIVO will be updated to remove whatever value is currently in VIVO. If you have None in a cell for the phone number of a person, then whatever phone number is currently in VIVO will be removed.
3. Your cell is empty or blank. Blank is also a special value for Simple VIVO. It means "do nothing." Whatever value VIVO might have for the attribute is left unchanged.

Using these three things, you can manage data in your VIVO using a spreadsheet.

An Example

Suppose we have data in our VIVO on our faculty. Three of the faculty members are shown below:

URI	Name	Phone Number
http://my.school.edu/individual/n123321	Jones, Catherine	345-8999
http://my.school.edu/individual/n467823	Pinckey, William	(404) 345-8991
http://my.school.edu/individual/n858832	Hernandez, Juan	

In reviewing this data, we find we would like to make the following changes:

1. Catherine's phone number should have an area code
2. William's last name is misspelled and should not have a "c"
3. William has left the university and his phone number should be removed
4. Juan's phone number should be provided.

We prepare an update set of data containing the following entries:

URI	Name	Phone Number
http://my.school.edu/individual/n123321		(404) 345-8999
http://my.school.edu/individual/n467823	Pinkey, William	None
http://my.school.edu/individual/n858832	Hernandez, Juan	(404) 345-8993

This update spreadsheet can be considered a set of instructions to Simple VIVO. We can read it say the following:

1. For the individual with URI http://my.school.edu/individual/n123321, leave the name unchanged (it is blank and blank means "do nothing") and change the phone number to (404) 345-8999.
2. For the individual with URI http://my.school.edu/individual/n467823, change the name

as shown and remove the phone number (None means remove whatever value you find)

3. For the individual with URI http://my.school.edu/individual/n858832, change the name as shown (Simple VIVO will notice that the name you gave is the name that is already in VIVO and so no change will actually be made. This is very handy. It means you can get data from VIVO and change only the items that need improving. All the other items can be left as they came from VIVO. The update will leave them untouched) and change the phone number to (404) 345-8993.

When the update is performed, the data in VIVO will look like:

URI	Name	Phone Number
http://my.school.edu/individual/n123321	Jones, Catherine	(404) 345-8999
http://my.school.edu/individual/n467823	Pinkey, William	
http://my.school.edu/individual/n858832	Hernandez, Juan	(404) 345-8993

Everything looks good. The names and phone numbers are all correct.

Data In

Getting data into VIVO is simple using Simple VIVO. You put the data you would like to add in a spreadsheet and save it as a "CSV File" -- a comma separated value file. You can also use a tab separated file (TSV) or use any
delimiter of your choosing (specified in the configuration parameters of Simple VIVO. For now, let's assume your
configuration is using a comma separated file.

You tell Simple VIVO the name of your spreadsheet and the name of your definition file. That's it. Simple VIVO
will add the data in your spreadsheet to the data in VIVO.

Let's say you wanted to add some new publications to your VIVO and your publications are stored in a spreadsheet called pubs.csv. Let's further assume that you have a definition file that defines your spreadsheet -- many such definition files are provided with Simple VIVO -- see [[Provided Data Scenarios|simple-vivo#provided-data-scenarios]] below. You would use:

```
python sv.py -a update -defn pubs_def.json -src pubs.csv
```

The data in `pubs.csv` will now be in VIVO. Simple.

Let's look at the items on the command line to see what each signifies:

`python sv.py` is the way we tell our system to run the Simple VIVO program. Your system administrator can confirm that this will work on your system.

`-a update` tells Simple VIVO that you want to update VIVO, using data from a source spreadsheet according to a definition.

`-defn pubs_def.json` tells Simple VIVO that you have a definition file called `pubs_def.json` that you would like to use to define the columns in your spreadsheet for the purpose of updating VIVO.

`-src pubs.csv` tells Simple VIVO that you have a data file called `pubs.csv` that will provide values to be added to VIVO.

Data Out

Getting data out of VIVO is simple using Simple VIVO. You specify the name of a definition file and the name of the file you want to store the data in. Simple VIVO uses the definition file to access your VIVO, retrieve the data,
and return a spreadsheet.

To retrieve a list of the faculty at your institution, you would use:

```
python sv.py -a get -defn faculty_def.json -src faculty.csv
```

The result is a new file called `faculty.csv` that contains the data for the faculty as defined by `faculty_def.json`

Each of the items on the command line are described below:

`python sv.py` is the way in which a python program such as `sv.py` is executed on your system. Your system administrator can determine if this is how you will start Simple VIVO on your system.

`-a get` indicates to Simple VIVO that you want to get data from VIVO. `-a` stands for "action". So you are requesting the `get` action.

`-defn faculty_def.json` indicates to Simple VIVO that you will be using the `faculty_def.json` definition file. This file must be available on your system. If Simple VIVO can not find the definition file you specify, it will
provide an error message to that effect.

`-src faculty.csv` indicates to Simple VIVO that the output spreadsheet will be called `faculty.csv`. Once Simple VIVO runs, you will have a new file called `faculty.csv`. You will be able to open the spreadsheet in Excel, Numbers, a text editor or other program.

Round Tripping

Getting data into VIVO and getting data out look very similar. The only difference is the action. The same definition file is used to get the data and to update the data. This insures that a spreadsheet that was produced by a "get" can be used by an "update".

It is common to get data from VIVO, improve it in some way, and then provide it back to VIVO. Suppose we were managing the organizations of our institution. We may have noticed that several of the institutions in our VIVO were missing URLs to their web pages, or missing phone numbers. We could use the steps below to improve the organizational data in our VIVO:

1. Get the organizational data from VIVO:

   `python sv.py -a get -defn org_def.json -src org.csv`

2. Open `orgs.csv` in Excel or your favorite spreadsheet editor and make changes as needed -- provide missing phone numbers, URLs or making other changes such as correcting spelling in names or organizations.

3. Put the improved data back in VIVO:

   `python sv.py -a update -defn org_def.json -src org.csv`

These steps are very common -- we get data from VIVO, improve it, and put the improved data back in VIVO.

Start at the Beginning

In VIVO, many things refer to other things. Publications refer to people as authors. Publications have dates, which in turn have facts about them. Publications refer to journals. To put in publications, we need to have the journals in first, as well as the people and the dates. In this way, the publication can refer to the people and to the dates that are already in VIVO.

The diagram above shows a simple rendering of these referrals. Things lower in the diagram have less dependencies and should be put in first. Things at the top of the diagram have more referrals. To put these things in VIVO requires that you have other things in VIVO first.

The Provided Data Scenarios (see below), build on each others. Dates is a very simple scenario with no references. Putting dates in first will lead to putting on Organizations, and so on.

Provided Data Scenarios

Follow one of the provided scenarios to manage data in your VIVO. The most common scenarios for managing data regarding the scholarship of your institution are provided.

- Dates
- Organizations
- Concepts
- Journals
- Locations
- People
- Positions
- Educational Background
- Awards and Honors
- Publications
- Grants
- Teaching
- Mentoring
- Memberships
- Editorial Activities
- Service

Adding Scenarios

Adding new scenarios requires a knowledge of the VIVO ontologies, as well as knowledge of the Pump Definition File. Studying the provided scenarios and the descriptions provided in this wiki will get you started. You will find support on-line on the VIVO email lists and in the VIVO Wiki.

Data Management

Data management a collection of processes and permissions and talents applied. To the task of creating quality data. Quality data is data is data that is correct and of value to consumers of the data.

Focus effort on high value data

VIVO represents the scholarship of your organization. It can represent a very broad collection of various kinds of activities of scholars -- the production of scholarly works, the acquisition of research funding, the teaching and mentoring of trainees, service to the profession, as well as details about the training and background of each person participating in the scholarship of the organization. Simple VIVO, as described in this book, focuses the collection and management of data on attributes of scholars that have high value for organizations. This focus enables us to manage data with a reasonable amount of effort.

Other choices could be made. simple VIVO is extensible. It can be used to manage any VIVO data, and actually, any semantic data. The data need not be the data described in the examples provided with Simple VIVO. See Extending Simple VIVO and the Pump descriptions of the definition file to see how Simple VIVO. can be extended to manage additional or different attributes regarding scholarship or other domains.

Authoritative Sources of Data

For every value in VIVO we should be trying to answer the question "where is the best source of this data?" For different examples, the answers are likely to be different. See the table below for possible answers. The answers for your organization may be different.

Example	Best source
People	Human Resources
Positions	Human Resources
Educational Background	The Provosts office
Grants	Office of Research
Teaching	The Registrar
Mentoring	Graduate School
Awards and Honors	Individual faculty members
Service	Individual Individual faculty members
Research areas	Individual faculty members
Publications	Publishers and indexers
Books	Individual faculty members

Many offices of your university may be best sources of information for VIVO. Perhaps their information has already been extracted and made available through a data warehouse. You may be able to use the data warehouse to provide data for your VIVO. In other cases, the data may need to come from the office to you. This may be able to done done in the form of a spreadsheet that can serve as input to you Simple VIVO processes.

Notice that some of the "best sources" are the faculty themselves. How I'll you collect this information from the faculty? Will faculty edit their information in VIVO? How will you follow up with faculty that do not provide information?

Foundational entities in VIVO may have data provided by VIVO or from other sources.

Example	Source
Dates	Generated by Simple VIVO from a range you provide
Locations	VIVO provides countries and US states. Simple VIVO provides US cities. Your organization provides locations of its campuses, and names of its buildings
Journals	Simple VIVO provides a list of 6,600 journals
Organizations	Simple VIVO provides a list of 2,500 universities in the US
Concepts	VIVO provides access to several controlled vocabularies

For each of the examples provided with Simple VIVO, we will assume that you have the data you need. We understand that getting the data from the various authoritative sources may require effort.

Values -- correct, missing, and incorrect

In Simple VIVO, whatever value you provide in the spreadsheet will replace the value in VIVO. Simple VIVO always assumes that the value in the spreadsheet is correct, or preferred, or better than the value in VIVO. When the value in the spreadsheet is the same as the value in VIVO, no work is done, the value in VIVO remains unchanged. When VIVO does not have a value, the value in the spreadsheet is added to VIVO.

In some cases, we may learn that the value in VIVO should be removed and should not be replaced with another value. VIVO may have a value that is incorrect, or it may have. A value that for some reason, should not be present in VIVO. Simple VIVO provides a special value "None" that is used in Simple VIVO to indicate that the value in VIVO should be removed and should not be replaced with another value. data managers sometimes. Refer to such values (or actually the lack of a value) as a missing value. To indicate a missing value, use the special value "None". If VIVO does not have a value, and the value. In the spreadsheet is None, no work is done and VIVO remains without a value. If VIVO has a value, and the value in the spreadsheet is None, the value in VIVO is removed.

In some cases, data you receive from authoritative sources may be incorrect. You will need to decide how to address such cases if they should arise. For example, you may receive data regarding the phone numbers of the faculty only to find that phone numbers do not contain the required number of digits, or have other validity problems. You may know of "better" phone numbers for one or more faculty. VIVO data managers often discover such concerns. It is customary to report the problems to the data provider.

In addition to providing values which replace values in VIVO, and providing the special value None to indicate that a value in VIVO should be removed, simple VIVO has a third important way of indicating what should be done as spreadsheet values are considered for inclusion in VIVO. A spreadsheet value can be blank or empty. When spreadsheet values are blank or empty, Simple VIVO interprets this as "do nothing". All blank and empty spreadsheet values are ignored during processing.

Binary values

Some values in Simple VIVO are defined as being binary -- that is, the value represented by the column is either present or absent in VIVO. Use a "1" in a binary column to indicate the presence of the value. Use "0" to indicate the absence of the value. "None" continues. To indicate that no value should be recorded. And blank continues to mean that no action should be taken.

Sets of Values

When managing data in VIVO, we need to. Be clear whether an attribute can have one value, or more than one value. VIVO often supports more than one value, but Simple VIVO may simplify the data management to the case of one value. For each example, we indicate the values that can be managed and whether the value is single, multiple, or binary (present or absent).

When Simple VIVO accepts multiple values for a value, put all the values in a single spreadsheet cell, separated by the "infra field delimiter" as specified in the `sv.cfg` file and defaulted to a semicolon.

As always, the values you specify in the spreadsheet will replace the values in VIVO. If you use the word None in a multi-valued field, all the values in VIVO will be removed. Leaving a multi-valued field blank indicates that no action should be taken on the field.

Summary

You will need to identify sources for each of the domains you choose to manage in Simple VIVO as well as procedures for acquiring the data, getting the data. Into the format required by Simple VIVO, and handling cases where the authoritative data may be incorrect.

Simple VIVO spreadsheet cells can contain any of the following

Value	meaning
A single value	Replace the value in VIVO with the value in the spreadsheet
None	remove the value in VIVO and do not replace it
Blank	do nothing
Multiple values separated by semi-colons	Replace the values in a multi-valued set with the values. In the spreadsheet
0	for a binary column, remove the value represented by the column
1	for a binary column, add the value represented by the column

Adding Dates to VIVO

Dates in VIVO are just like other things in VIVO -- they have URI and they have attributes. Unlike other things, however, VIVO creates the same date many times, giving each its own URI. Eventually your VIVO will have the same date represented many, many times.

Using Simple VIVO, we can create a set of dates for VIVO that are available for reuse in managing all other entities.

For example, when entering a publication in VIVO, one is asked to enter the date of the publication. Rather than create a new date for each publication, we can create a set of dates that are reused for *all* publications. This has several advantages over creating dates for each publication:

1. It avoids semantic confusion. Having two dates that represent the same thing (the same date) but have different URI is not a best practice. We could assign "sameAs" predicates to equate all the various entities representing the same date, but this just layers complexity on an unfortunate situation.
2. It is simpler. There will be only one "December 18, 1993." All things that need to refer to this date will refer to the one entity in VIVO representing "December 18, 1993."
3. It saves space. As VIVO grows and adds publications, it adds dates. But this is unnecessary. All the dates VIVO needs can be created and loaded into VIVO once and then reused by all other data management practices.
4. It supports look-up by date. Rather than find all the entities with the same date value and then finding all the things associated with each of those values, we can find all things associated with the single entity that represents the date of interest.

Date Precision

VIVO supports the concept of "date precision." This is a little unusual in the world of data management, so let's take a look at what VIVO can represent. VIVO provides three precisions for date values. A date can be just a year, or just a year and a month (common in publication dating) or a year and a month and a day (common in most administrative function). In some cases, systems other than VIVO are "required" by their internal formats to provide a day, month and year for all dates. Many systems default the unknown components of the date to "1". This means that when looking at a date (without a precision) of "1993-01-01" we are unable to discern whether the date value is informative about the month and day, or whether these are just place holders. VIVO is capable of representing, with certainty, the level of precision in the date value. The date value "1993-01-01" with a date precision of "http://vivoweb.org/ontology/core#yearPrecision" is the same as a date value of "1993" with the same year precision. The month and day values are non-informative.

VIVO date precision	Example	Since Jan 1, 1800 until Dec 31, 2050
http://vivoweb.org/ontology/core#yearMonthDayPrecision	"1993-12-18"	91,676
http://vivoweb.org/ontology/core#yearMonthPrecision	"1993-12"	3,012
http://vivoweb.org/ontology/core#yearPrecision	"1993"	251

Pre-loading VIVO with dates

We can pre-load VIVO with dates using the dates example in Simple VIVO. We can set a range of dates that will be useful for our VIVO, generate them and update our VIVO with them. Subsequent updates can use the dates that are already in VIVO, rather than making more.

examples/dates

The dates folder in examples has everything we need to pre-load our VIVO with dates.

`gen_dates.py` is an optional Python program for generating a spreadsheet of dates data. By default it generates dates from Jan 1, 1800 to Dec 31, 2050. This should be more than enough for many VIVOs. Dates are generated for each year, for each year/month combination and for each date in the range. By modifying `gen_dates.py` it can produce dates in any range.

`dates.txt` is a tab delimited spreadsheet ready for the pump.

`sv.cfg` is a config file that defines the dates update. You will need to edit this to supply query parameters for your site. Your system administrator will be able to supply you with appropriate query parameters.

`date_def.json` is a definition file for putting dates in VIVO. It defines three columns: 1) uri, 2) Precision, and 3) Date.

`datetime_precision_enum.txt` is an enumeration of the VIVO datetime precisions that allow the use of short codes "y", "ym" and "ymd" for the precisions described above.

To run the example, use:

```
python sv.py
```

The result will be two files: `date_add.rdf` and `date_sub.rdf` Use the VIVO system administration interface to add `date_add.rdf` to your VIVO. The result will be 94,939 dates. You may never have to be concerned about dates again.

Managing Journals in Simple VIVO

Journals are a foundational entity for VIVO. To add publications to VIVO using Simple VIVO, the journals for the publications must already be present in your VIVO. You will want to load your VIVO with journal titles and add journals as required for your publications.

Adding journals to your VIVO

Simple VIVO provides a set of more than 6,600 academic journals, each with an ISSN. Many also have an EISSN.

To add these journals to your VIVO, use:

```
python sv.py -a update -s journal_data.txt
```

The result will be a two files: `journal_add.rdf` and `journal_sub.rdf` Use the VIVO System Admin interface to add the RDF in `journal_add.rdf` to your VIVO. `journal_sb.rdf` should be empty. If not, use the VIVO System Admin interface to subtract the RDF in `journal_sub.rdf` from your VIVO.

Managing Journals in your VIVO

1. Use get to retrieve the journals from your VIVO to a spreadsheet

   `python sv.py -a get`

2. Edit your spreadsheet to improve your journal data, and add new grants. You may need to remove duplicates, correct spelling or otherwise improve the name of the journal, or add an EISSN if one is missing.

3. Use update to put your improved journal data back in VIVO

   `python sv.py -a update`

4. Repeat the steps each time you wish to improve your journal data

Managing Locations

VIVO can store information about locations such as cities, buildings, and campuses. By providing latitude and longitude information for your locations you will be able to make maps and show information based on location.

One Simple VIVO definition can be used to manage all your locations, or you may choose to use source files specific to different kinds of locations -- one for cities, one for buildings, one for campuses.

Many types of things are locations. The most common are campuses, buildings, countries, and cities, but see the file `examples/locations/location_types.txt` for a complete list of VIVO location types.

Regardless of the type of location, each location has:

1. a name
2. one or more types such as building, campus, country or city
3. within -- an indication that one location is within another. For example, a city is within a country, a building is within a campus.
4. latitude. Negative is south.
5. longitude. Negative is west.

We can use one Simple VIVO scenario to manage all of our locations.

1. See `locations.txt` for sample locations.
2. Using a spreadsheet, edit this file to remove locations that you do not wish to have in your VIVO.
3. Add locations of interest for your VIVO.
4. Run `python sv.py -a update` to put locations in your VIVO.
5. Use `python sv.py -a get` to get a spreadsheet of all the locations currently in your VIVO.

That's it.

Two additional examples are provided. Buildings can be managed using the methods described below. You may find that having buildings in their own spreadsheet makes working with them easier than having all your locations in a single spreadsheet.

A second example provides a data set of United States cities with populations of over 100,000. This data set can be loaded into your VIVO and used to provide references for events and other VIVO entities that reference locations.

Manage Campus Buildings using Simple VIVO

Buildings are locations. Each has a name, a URL (may point to an enterprise system, an existing university web page about the building, historical information, or any other information resource about the building), a latitude and longitude.

Note

A sample file of buildings at the University of Florida is provided in uf_buildings.txt

How to use

1. Use the sv_buildings.cfg config file to get your buildings from VIVO, as in: `python sv.py -c sv_buildings.cfg -a get`
2. Edit the resulting buildings.txt file to add your buildings
3. Update VIVO using `python sv.py -c sv_buildings.cfg -a update`
4. Add the resulting RDF to VIVO

Add US Cities to VIVO

Add cities in the United States with population over 100,000 in 2015 to VIVO. The list came from Wikipedia, updated with cities over 100,000 in Puerto Rico.

Each city has a name, a state, a latitude and a longitude.

Look up the state in VIVO through an enum. If not found, throw an error -- all the states need to be in VIVO. VIVO is missing District of Columbia and Puerto Rico. These need to be added by hand before the cities can be added.

Look up the city in VIVO. Update the state, lat and long as needed.

Note

There's nothing US specific about this code. The data file contains US cities. Edit the data file to add cities of interest in provinces or states of interest.

The def file assumes that the wgs ontology has been added into your VIVO. See World Geodetic System in Wikipedia and Basic Geo Vocabulary and the RDF file

How to use

1. Add District of Columbia to your VIVO as a State or Province. Add to state_enum.txt
2. Add Puerto Rico to your VIVO as a State or Province. Add to state_enum.txt
3. Add the cities to your VIVO using an sv update as shown `python sv.py -c sv_cities.cfg -s us_cities.txt -a update`
4. Add the resulting RDF to VIVO
5. To see your city data and manage it in the future, you can do an sv get: `` `python sv.py -c sv_cities.cfg -a get ``

Managing Organizations

You will want to have organizations in your VIVO. Here are some reasons:

1. Your colleges, academic departments, administrative units, and laboratories are organizations in VIVO. VIVO can represent the organizational chart of your institution.
2. Organizations are where your people had positions before being at your institution.
3. Your people received degrees from organizations.
4. Organizations provided grant funding to your organization.
5. Your faculty have collaborators at other organizations.
6. Organizations are employers of your graduates.

Reports and visualizations use organizational data.

Create a spreadsheet of the organizations that you can use to manage the organizational data. As organizations come and go, add and remove them from your spreadsheet. As contact information changes, update your spreadsheet. Use Simple VIVO to update VIVO using the values in your spreadsheet (see below).

Recall that Simple VIVO always allows three things to appear in a cell in a spreadsheet cell:

1. The value of the attribute. See below. The cell may contain a name, or a phone number, or a zip code. The cell may contain a delimited list of types. The cell value will replace the value in VIVO.
2. The word None. None directs Simple VIVO to remove whatever value it finds in VIVO.
3. An empty cell, that is, nothing is in the cell. When Simple VIVO sees nothing in the cell, it does nothing -- it neither adds nor subtracts a value.

Simple VIVO ignores all empty cells. Simple VIVO compares non-empty cells to values in VIVO. If the value in the cell is different than the value in VIVO, Simple VIVO replaces the value in VIVO with the value in the cell. If the value in the cell is the same as the value in VIVO, Simple VIVO does nothing. Cells with the word None act to erase values from VIVO.

examples/orgs

examples/orgs contains the files you need to manage organizations using Simple VIVO.

Your spreadsheet data should have the columns shown in the table below. `org_def.json` defines these columns. Your VIVO site administrator should be able to help you edit `org_def.json` to add or remove columns.

Column name	Purpose
uri	The VIVO uri of the organization. VIVO assigns a uri to every organization in VIVO.
remove	Used to remove organizations from your VIVO.
name	The name of the organization precisely as it will appear in VIVO
home_page	the URL of the home page of the organization
email	The email address of the organization
phone	The phone number of the organization
within	The name of another organization which is the parent of this one
address1	Address line 1 for the organization
address2	Address line 2 for the organization
city	The name of the city of the organization
state	The name of the state of the organization
zip	the sip code (postal code) of the organization
abbreviation	An abbreviation for the organization. "NIH" for the National Institutes of Health, for example
overview	A paragraph of text describing the organization
type	The organization's types. See below. Organizations may have more than one type.
successor	The name of the successor to the organization if the organization no longer exists

Creating `org_enum.txt`

Organizations have "parent" organizations. The Chemistry Department may have the parent "College of Liberal Arts and Sciences. The College may have the university as a parent. It is easy to record this information in your spreadsheet. Use the "within" column to name the parent of the organization. Simple VIVO will compare the name you provide to a list of names in `org_enum.txt`. Simple VIVO will then use the URI of the parent.

To make `org_enum.txt`, run `make_enum.py` using:

```
python make_enum.py
```

Update `org_enum.txt` each time you add organizations to your VIVO.

Updating organizations in VIVO

1. Get your organizations from VIVO using `python sv.py -a get` Simple VIVO will create `orgs.txt`
2. Edit `orgs.txt` using a text editor or a spreadsheet. The file is delimited using the inter field delimiter, which defaults to the tab character.
3. Update your organizations using ``` `python sv.py -a update ``` Simple VIVO will create `org_add.rdf` and `org_sub.rdf`
4. Add `org_add.rdf` to VIVO using the Site Admin Load RDF feature
5. Remove `org_sub.rdf` from VIVO using the Site Admin Remove RDF feature
6. Update the enumeration using `python make_enum.py`
7. Repeat these steps to make changes to your organizational data

Adding organizations to VIVO

Adding orgs is simple -- just edit your spreadsheet to add a row. Put as much data about the organization as you have on the row in the appropriate columns. Leave the rest of the columns blank. In particular, leave the `uri` column blank. Simple VIVO will assign a uri to your organization when it is added.

Follow the rest of the steps described for updating your organization values. Adds and updates can be combined in the same spreadsheet.

Removing orgs from VIVO

Removing orgs is also very simple -- just edit your spreadsheet and put the word "remove" in the `action` column. Each organization that has `remove` in the action column will be removed from VIVO.

Maintaining subsets of orgs

You may find that you would like to have different kinds of organizations in different spreadsheets to simplify data management. For example, you may wish to have your internal organizations in one spreadsheet and external organizations in another.

To maintain subsets, follow the steps below:

1. Create appropriate folders for your work. You may want to manage the two collections from two different folders.

2. Copy the def file into each folder
3. Modify the def file to select the orgs of interest. `entity_def` defines the rows of the spreadsheet. Write a query to select the rows you want in your subset.
4. Copy the `sv.cfg` file into each folder
5. Modify the `sv.cfg` file to use the name of the definition file for your subset. Modify the rdfprefix and source file name as appropriate to help you manage the subset.
6. Run your gets and updates as above in each folder for each subset.

Frequently Asked Questions

Do I need to have all the possible columns in my spreadsheet?

No. The only columns you need are the uri column, and the columns you want to update.

Do I need to have all the organizations in my spreadsheet?

No. The only rows you need are for the organizations you want to update.

So if I wanted to update email addresses for the departments in my college all I would need is the names of the departments and their email addresses?

Yes.

But sometimes two organizations might have the same name -- the chemistry department at my school and the chemistry department at Berkeley, for example. How do I handle that?

There are two approaches.

The first is simple -- include the name of the school in the name of the org for any org outside your institution. So, for example, at your school, you would name the chemistry department "Chemistry" and the chemistry department at Berkeley would be called "Chemistry (UC Berkeley)"

The second approach is more complex, but will allow you to avoid having to put the name of the school in the name of the organization. When using Simple VIVO, the spreadsheet provided by Simple VIVO contains an additional column that you did not enter. That column is called "uri." VIVO supplies a uri for each thing in VIVO. Every org has a uri. If you add a second Chemistry department with no URI, VIVO will add it and give it a new uri. You will then have two departments, both named Chemistry, but with different URIs. If one has a parent at your univesity and the other has a parent at the other university, you will be able to identify each. You must modify `make_enum.py` It assumes that an organization can be identified by a unique name.

What if I use a column name in my spreadsheet that is not in the def file?

Nothing will be done. The log will contain messages indicating which columns were unknown. Always check the log after every run to see what Simple VIVO has done. Only column names that are in your spreadsheet **and** in the definition file are processed. All other columns are ignored.

For a `get` this means that Simple VIVO will get all the columns defined in the definition file and return them as columns in a spreadsheet.

For an `update` this means that if a column appears in your spreadsheet that is not defined in the definition file, it will be ignored during the update. You can have a "notes" column n your spreadsheet, for example, where you leave notes to yourself regarding additional research that should be done regarding a particular entity.

For an `update` in which all the columns in the spreadsheet are defined, all the columns in the spreadsheet will be processed. If there are additional column definitions in the definition file and those columns are not in your spreadsheet, these additional definitions will not be used. This makes it easy to have a definition file that defines many columns and then only use a few columns in a particular spreadsheet. You might have a spreadsheet that updates organizational contact information and a separate spreadsheet that updates `within` and `successor` information, for example.

What if I give a new organization a name -- one that VIVO hasn't seen before?

Simple VIVO will add the organization to VIVO and give it all the data you have on the the row in your spreadsheet. It will assign a new URI. The next time you do a `get` , Simple VIVO will show all your organizations, including the ones you added. Be sure to run `make_enum.py` to update `org_enum.txt`

How do I remove an organization?

Create a spreadsheet with two columns. In the uri column, give the URI of the organization to be removed. Create a second column with the column heading "action". On the row under action, put "remove".

How do I remove values for some organizations and not others?

Use "None" to remove a value from an organization.

What happens if I leave the value in a column for an organization blank?

Nothing happens. This is different from specifying a value, or specifying "None" to remove a value. Three things can appear in a column and Simple VIVO takes three different actions:

- If a value appears in the column, the value in VIVO is replaced with the value you specify.
- If the value "None" appears in the column, Simple VIVO removes the value in VIVO.

The org in VIVO will have no value for the column containing "None".

- If the value is blank, Simple VIVO does nothing. The value in VIVO, if there is one, remains unchanged.

Managing Concepts Using Simple VIVO

One of the great advantages of VIVO is its ability to use controlled vocabularies to manage concepts across domains. Your grants, papers, fields of study, personal research areas and others can all be coded using vocabularies that are appropriate for the job.

Managing concepts in Simple VIVO is straightforward. You edit concepts in `concepts.txt` as a spreadsheet. You use `update` to update VIVO. You use `get` to retrieve concepts from your VIVO.

Get started

Run `make_enum.txt` to create an enumeration of the concepts in your VIVO. This will allow you to use concept labels in your spreadsheet to refer to concepts that are "narrower" or "broader" than the concept you are editing.

```
python make_enum.txt
```

Get Concepts from VIVO

Use

```
python sv.py -a get
```

Edit your Concepts

Use a spreadsheet program such as Excel or Numbers to open `concepts.txt` Edit and save.

Update your VIVO

Use

```
python sv.py -a update
```

Each time you wish to update your concepts -- add new concepts, add narrower or broader concepts to existing concepts, change the labels of concepts -- you can repeat the steps below:

1. Update the enumerations using `make_enum`
2. Get the concepts from your VIVO into a spreadsheet using `sv.py -a get`
3. Edit the concepts using a spreadsheet program
4. Update VIVO using `sv.py -a update`

Managing People in VIVO

Each person in VIVO has contact information and other identifying information as expected.

This Simple VIVO example supports 19 attributes for each person. Each one is optional.

1. display_name
2. orcid -- enter the person's orcid id in the form nnnn-nnnn-nnnn-nnnn
3. types -- enter the person's type(s) using the abbreviations in the `person_types.txt` enumeration.
   `fac` for faculty, `pd` for postdoc, etc.
4. research_areas -- enter the person's research area(s) using the
5. overview -- enter a plain text overview for the person. Do not cut and paste from Microsoft Word.
6. name_prefix -- name prefix such as "Dr"
7. first_name -- person's first name
8. middle_name -- person's middle name
9. last_name -- person's last name
10. name_suffix -- person's name suffix such as "Jr" or "III"
11. title -- person's title
12. phone -- person's primary phone number
13. email -- person's primary email
14. home_page -- person's home page URL
15. street_address -- mailing address street for the person
16. city -- mailing address city
17. state -- mailing address state or province
18. zip -- mailing address postal code
19. country -- mailing address country

As with everything in Simple VIVO, the attributes are optional. For any particular person, enter as many attribute values as you have. It is straightforward to update the person with additional attributes in the future.

Enumerations

1. `concept_enum.txt` -- used to list possible research areas for the person
2. `person_types.txt` -- used to list possible person types with abbreviations for each

Adding and Updating people

1. Update the enumerations using `python make_enum.py`
2. Get the existing people from VIVO using `python sv.py -a get`
3. Edit the people as needed -- correcting values as needed, adding values to people in the spreadsheet, and adding new people. To add new a new person, add a row to the spreadsheet, leave the uri column blank, and add as many attributes as you have for the person. All the attributes you add should be found in the appropriate enumerations (see above). If the value you specify is not found, it will not be added. To add new values to enumerations, first add them to VIVO, then update the enumerations, then add them to the person.
4. Update VIVO with your improved spreadsheet using `python -a update`

Repeat these steps as needed to manage the people in your VIVO.

Managing Positions Using Simple VIVO

People have positions in organizations. Some positions are "official" -- a person is employed with a job title such as "Professor" or "Research Assistant." Some positions may be "official" but unpaid -- a person may be director of a center or president of a professional society. Some positions may be unofficial -- a person may be an evaluation lead for a program. All of these can be represented using VIVO positions using the techniques described below.

Positions have five attributes:

1. Start date -- from an enumeration of dates of the form yyyy-mm-dd
2. End Date -- from an enumeration of dates of the form yyyy-mm-dd
3. Organization of Position -- from an enumeration of organizations by name
4. Person in position -- from an enumeration of people by orcid
5. Job title -- open text

The steps to manage positions

1. Update your enumerations using `python make_enum.py`
2. Get positions from your VIVO using `python sv.py -a get`
3. Edit your spreadsheet `positions.txt` correcting and adding information for existing positions, and adding new positions. Use a text editor or spreadsheet program
4. Update the position information in your VIVO using `python sv.py -a update`

Repeat the steps as necessary.

Managing Educational Background

Simple VIVO can be used to represent the educational backgrounds of your people. Each person can have one or degrees. These degrees will be displayed on profile pages and can be used in queries.

To add educational backgrounds to VIVO, we need to know several pieces of information for each degree:

1. Who was awarded the degree? This person must already be in your VIVO. See the People example in Simple VIVO regarding managing people in your VIVO.
2. What degree was obtained? For example, PhD, BA, MS, etc. These degrees must already be in your VIVO. VIVO is delivered with more than 100 common degrees. If you need to add more, you can use the System Admin web interface.
3. From what institution? The institution awarding the degree must already be in your VIVO. See Organizations for examples of adding organizations to your VIVO.
4. What is the date of the degree? The date must already in in your VIVO. See the Dates example in Simple VIVO.
5. What is the field of study for the degree? The field of study must be listed in the `field_enum.txt` file provided in the `education` folder of Simple VIVO. More than 800 common fields of study are provided. Edit this file using a text editor to add additional fields of study.

Each of these attributes is optional.

Edit the Configuration File

Edit the `sv.cfg` file to provide the query parameters for your VIVO. Your system administrator can help you with this.

Managing Enumerations

Each attribute of the degree is represented by an enumeration. This guarantees that all the information regarding education in VIVO is "coded" -- there are no open-ended text fields associated with educational background, and "linked" -- we can easily find all the people with a particular degree, all the people who have degrees from particular institutions, and all other combinations of attributes. If a value for one of the attributes can not be found in the enumerations, it will need to be added before the degree will be added to VIVO.

Five enumerations are used for the five attributes:

1. `person_enum.txt` for who
2. `degree_enum.txt` for what
3. `school_enum.txt` for where
4. `date_enum.txt` for when
5. `field_enum.txt` for field of study

The python program `make_enum.py` can be used to update four of these enumerations from your VIVO. To run `make_enum.py` use:

```
python make_enum.py
```

Fields of Study

The field of study enumeration is managed by hand -- if you have fields of study that you would like to be able to represent in VIVO, edit the `field_enum.txt` file with a text editor and add a row for each such field of study. If there are fields of study in the provided enumeration that you do not wish to allow ion your VIVO, remove these rows from the enumeration.

You will notice that the short form of the field of study and the VIVO form are identical. Field of study is a text field, but one which we would like to control -- that is, we do not want arbitrary text to appear in the fields of study. The field of study enumeration is used to look up each attempt to add a filed of study value, and confirm that the field of study is on the list. Any list of fields of study can be controlled in this manner.

Procedure

1. Get the existing educational backgrounds from your VIVO into a spreadsheet. To do this, run get: `python sv.py -a get` The result will be a `degrees.txt` file.
2. Edit your enumerations as needed to provide additional values for degrees and fields of study. To add universities or other schools to your VIVO and make them available for your education work, use the orgs example to add the organizations, then run `make_enum` to update your enumerations. The new schools are now available for putting educational backgrounds in VIVO.
3. Edit `degrees.txt` in a spreadsheet program to update the values of existing degrees, or add new degrees -- one degree per row. Every value you enter in each of the columns must be a value in one of the enumerations. Each column has its own enumeration.
4. Run update `python sv.py -a update` The information in your `degrees.txt` file will be used to update degree information in VIVO.

Managing Awards and Honors using Simple VIVO

For Awards and Honors, VIVO distinguishes between two ideas:

1. The generic award (as in the Nobel Peace Prize)
2. And the instance of the award as in "Theodore Roosevelt won the Nobel Peace Prize in 1906"

This allows us to easily list all the people who won any particular award.

To add awards and honors to VIVO, we need to know several pieces of information for each award or honor:

1. Who got the award or honor? e.g. Theodore Roosevelt
2. What award or honor was received? e.g. Nobel Peace Prize
3. From what organization? e.g. Norwegian Nobel Committee
4. What is the date of the award? e.g. 1906

Each of these four things "Theodore Roosevelt", "Nobel Peace Prize," "Norwegian Nobel Committee," and the date 1906 must all be in VIVO *before* the indication of the award to Roosevelt can be added by Simple VIVO. Adding Awards and Honors makes the links between these four things, not the things themselves.

Managing Enumerations

Each of the four things is represented by an enumeration:

1. person_enum for who
2. award_enum for what
3. org_enum for where
4. date_enum for when

The python program `make_enum.py` can be used to update these enumerations using data from your VIVO. To run `make_enum.py` use:

```
python make_enum.py
```

Managing Awards

Once the things to be talked about are in VIVO, and the enumerations are made, you can use

```
python sv.py -a get
```

to get the awards and honors from your VIVO into a spreadsheet called `awards.txt` You can add awards to your spreadsheet being sure to use values from each of your enumerations.

Once you have awards to add to VIVO, use:

```
python sv.py -a update
```

which will create the `award_add.rdf` and `award_sub.rdf` to update your VIVO.

Adding People, Awards, Dates, and Organizations to your VIVO

To manage awards, your enumerations must be up to date. Each person, organization, award, and date that will be referenced in an award must already be in your VIVO and in your enumerations. Updating your enumerations is simple -- run `make_enum.py` as shown above.

If you need to add dates, rerun the dates example with a wider date range.

To add a single person, name of an award, or an organization conferring an award, use the VIVO web interface. Update the enumerations, add a row to your `awards.txt` for the new award and run a Simple VIVO update.

For example:

John Smith earns a lifetime achievement award from the American Cancer Society in 2015. John Smith and the date 2015 are already in your VIVO, but the American Cancer Society and the lifetime achievement award are not.

1. Use the Site Admin interface to add the American Cancer Society as an organization.
2. Use the Site Admin interface to add "ACS Lifetime Achievement Award" as an award.
3. Update the enumerations. Now all four elements of the award receipt (person, date, award and organization) are in your VIVO and in your enumerations. You can add a row to your `awards.txt` spreadsheet that looks like

```
John Smith ACS Lifetime Achievement Award American Cancer Society 2015
```

4. Run `python ../../sv.py -a update` and the award will be in place.

Managing Publications in Simple VIVO

Is this the biggest topic? Has handlers for pulling data from PubMed, CrossRef and eventually SHARE.

Include the Harvard PubMed identifier software.

Managing Authors in Simple VIVO

Authors -- stubs, local authors, disambiguation. ORCID, other. Big topic.

Managing Grants in Your VIVO using Simple VIVO

Grants are a bit more complex than some of the other entities we have been managing using Simple VIVO. Grants have the following columns:

1. Local award ID -- that is, what number, or other identifier, does your institution give this grant
2. Title -- the title of the grant. This should be spelled out -- do not use abbreviations. There is no length limit.
3. Direct costs -- all years -- a number without a dollar sign and with decimals as needed. For example `43506.45`
4. Total award amount -- all years, direct plus indirect. Same format as direct costs.
5. Principal investigators -- one or more principal investigators. Investigators are specified using their orcid ID and separated by your intra field delimiter.
6. Co-principal investigators -- any number of co-principal investigators. Specified using orcid, separated by intra field delimiter.
7. Investigators -- any number of investigators. Specified using orcid, separated by intra field delimiter.
8. Concepts -- any number of concepts identifying subject areas for the grant. Concepts are specified by name as they appear in the `concept-enum.txt` file. Concepts are separated by the intra field delimiter
9. Start date in the form yyyy-mm-dd
10. End date in the form yyyy-mm-dd
11. Administering unit. Specified by name as they appear in `dept_enum.txt`
12. Sponsor. Specified by name as they appear in `sponsor_enum.txt`
13. Sponsor's award ID -- how does the sponsor refer to this grant. The National Institutes of Health in the US, for example, gives its grants award ID that look like R01DK22343
14. URI in VIVO. Blank if you are adding the grant. VIVO URI returned by get.

Enumerations

Simple VIVO for grants uses several enumerations to identify data in VIVO. Each enumeration is made by `make_enum.py` as described below.

1. `concept_enum.txt` lists the concepts in your VIVO by name
2. `date_enum.txt` lists the year-month-day dates in your VIVO by date value
3. `dept_enum.txt` lists your institutions organizations by name

4. `orcid_enum.txt` lists your investigators by ORCID. ORCID is used to identify your investigators.
   Each should have an ORCID, and the ORCID for each investigator should be in your VIVO. See examples/people regarding adding ORCID to people.
5. `sponsor_enum.txt` lists the sponsors (funding organizations) in your VIVO.

Each attribute is optional. You may add a grant with very little information and provide additional or improved values for attributes in subsequent updates.

As always, your spreadsheet values will be non-blank if you would like to specify a value for the attribute, blank if you would like to have VIVO remain unchanged, and "None" if you would like to remove the attribute's value from VIVO.

The Basic steps

1. Use `make_enum.py` to prepare the enumerations for grants

   ```
   python make_enum.py
   ```

2. Use get to retrieve the grants from your VIVO to a spreadsheet

   ```
   python sv.py -a get
   ```

3. Edit your spreadsheet to improve your grant data, and add new grants

4. Use update to put your improved grant data back in VIVO

   ```
   python sv.py -a update
   ```

5. Repeat the steps each time you wish to improve your grant data

Managing Teaching using Simple VIVO

To manage teaching in Simple VIVO, you will need to manage a list of courses, and you will need to specify teaching -- who taught what when.

Each is described below.

Courses

Managing a list of courses is easy. Courses have two columns -- the name of the course and the concepts of the course. If your courses have course numbers, you may want to include them in the name of the course. So, for example, if the name of your course is "Introduction to Statistics" and the course number is "STA 1201," you may want to have the name in VIVO be "STA 1201 Introduction to Statistics"

You can use:

```
python -a get -c sv_courses.cfg -s courses.txt
```

to get a spreadsheet of the courses you have in VIVO. Add the names of courses, or edit the names as needed, and then update your list in VIVO using:

```
python -a update -c sv_courses.cfg -s courses.txt
```

Teaching

To manage teaching, you will need a list of courses in VIVO (see above).

For each course taught by an instructor, you will need to know:

1. The course that was taught -- by name of the course
2. The person who taught the course -- by orcid
3. The start date for the course -- year month day by value
4. The end date for the course -- year month day by value. For example "2015-12-09"

You can use:

```
python make_enum.py
```

to prepare the enumerations for your work. Then use:

```
python sv.py -a get
```

to get a spreadsheet of your courses. Add to it, edit, remove as needed.

You can then use:

```
python sv.py -a update
```

to update VIVO to include your changes.

Repeat these four steps -- make_enum, get, edit, update -- whenever you would like to edit your teaching records.

Managing Mentoring Relationships using Simple VIVO

Mentoring relationships between mentors and mentees. Any context.

Managing Memberships

People have memberships in organizations. People may be chairs or members of committees, or have memberships in professional organizations. Memberships have date ranges. All of these can be represented by VIVO memberships using the techniques described below.

Memberships have five attributes:

1. Start date -- when did the membership start? From an enumeration of dates of the form yyyy-mm-dd
2. End Date -- when did the membership end? Leave blank if the membership is on-going. From an enumeration of dates of the form yyyy-mm-dd
3. Organization of Membership -- from an enumeration of organizations by name
4. Person in position -- from an enumeration of people by name
5. Membership role -- open text. Typically "Member" or "Chair."

The steps to manage memberships

1. Update your enumerations using `python make_enum.py`
2. Get memberships from your VIVO using `python sv.py -a get`
3. Edit your spreadsheet `membership.txt` correcting and adding information for existing memberships, and adding new memberships. Use a text editor or spreadsheet program
4. Update the membership information in your VIVO using `python sv.py -a update`

Repeat the steps as necessary.

Managing editorial activities in Simple VIVO

Editorial activities such as editor, associate editor, assistant editor, or reviewer of a journal have five attributes:

1. Start date for the activity -- from an enumeration of dates of the form yyyy-mm-dd
2. End date for the activity -- from an enumeration of dates of the form yyyy-mm-dd
3. Journal editorial activity -- from an enumeration of journals by issn
4. Person holding editorial activity -- from an enumeration of people by label
5. Editorial Role -- either "editor" (for all kinds of editorships) or "reviewer" for all kinds of reviewers.
6. Editorial Role label -- open text -- further describes the role as needed. Examples: "Editor in Chief" or "Applications Reviewer"

Simple VIVO will work with whatever data you have -- any of the attributes can be left blank. But an editorial activity without a journal or person or role is not expected by the users of VIVO.

The steps to manage editorial activities

1. Update your enumerations using `python make_enum.py`
2. Get editorial activities from your VIVO using `python sv.py -a get`
3. Edit your spreadsheet `editorial.txt` correcting and adding information for existing editorial activities, and adding new editorial activities, one row per editorial activities. If a person has five editorial activities, use five rows for that person. Use a text editor or spreadsheet program to edit `editorial.txt`
4. Update the editorial activity information in your VIVO using `python sv.py -a update`

Repeat the steps as necessary.

Managing Service to the Profession using Simple VIVO

Service to the profession. Positions in academic societies, editorships and reviewer positions for journals.

University of Florida Examples

The University of Florida (UF) uses the Pump and Simple VIVO to manage its enterprise VIVO. Both the Pump and Simple VIVO were developed at UF to meet the need of upgrading VIVO from the 1.5 ontology to versions of VIVO supporting the 1.6 ontology (VIVO software version 1.6 and above). The University had tens of thousands of lines of code that were dependent on the 1.5 ontology -- code that put data into VIVO and code that took data out of VIVO. These two basic functions became the update and get functions that we see in the Pump and Simple VIVO.

The University of Florida examples have several characteristics that distinguish them from the previous Simple VIVO examples:

1. The examples are large scale. UF is a large institution with over 900 institutional organizations, 6,000 faculty, and approximately 6,000 publications and 2,00 new grants per year.
2. The examples demonstrate the clean-up of enterprise data for display in VIVO. Like most institutions, the data of the University of Florida is optimized for internal transactional processing. It was never intended for integrated, external display of the scholarly work of the organization. Much effort has been put into preparing for integration and display of the scholarly work.
3. The examples use data source external to the organization. Examples show the use of PubMed and other external sources to augment the institution's data.
4. The examples make use of ontology extensions. UF finds it convenient to extend the ontology for several purposes -- to better represent its work, and to provide local identifiers for entities in VIVO.

The UF domains

1. Orgs
   - Organization data management -- management via spreadsheet. No institutional data
2. People
   - Ensure UF business rules
   - Manage UFCurrentEntity
   - Support excluding people from automated edited
3. Position Management
   - Support UF terminology and business rules

    - Exclude positions in protected departments
    - Improve position titles
    - Manage types
4. Papers
    - Ingest from PubMed IDs
    - Ingest from DOI
    - Ingest from CSV (bibTex to CSV)
5. Courses
    - Include common course number. Serve as the abstraction layer for course sections, much as awards have AwardReceipt and Award, and degrees have AwardedDegree and AcademicDegree, UF recognizes CourseSection (as taught) and Course (model).
6. Course Sections
    - Link to courses, instructors, terms
7. Sponsors
    - With UF sponsorid
8. Grants
    - Linked to departments, datetimes, cois, pis, invs

Versions of VIVO

Simple VIVO works using definitions files that "map" your spreadsheet rows and columns to VIVO's internal data models. VIVO's internal data models are defined using several "ontologies." As with software, ontologies have versions. Some changes in ontologies provide new capabilities for representing additional kinds of objects, or provide additional attributes for existing objects. But some ontology changes require changes to existing VIVO objects and the maps to them used by Simple VIVO.

The version of Simple VIVO described in this book is for use with VIVO 1.6 and above. These versions all use the same ontology (VIVO-ISF 1.6) and can all run the examples in this book.

At some point, VIVO will be updated to new versions of ontologies that require mapping changes. When that happens, the Simple VIVO software will be updated to reflect the VIVO changes as reflected in the ontology changes.

Simple VIVO Command Line

Here we describe the command line parameters and how each is used.

To see the available Simple VIVO command line paraemters and get a brief description of each one, go to the command line and type

```
python sv.py -h
```

You will see the display below.

```
Get or update row and column data from and to VIVO

optional arguments:
  -h, --help            show this help message and exit
  -a [ACTION], --action [ACTION]
                        desired action. get = get data from VIVO. update =
                        update VIVO data from a spreadsheet. summarize = show
                        def summary. serialize = serial version of the pump. test = te
st connection to VIVO.
  -d [DEFN], --defn [DEFN]
                        name of definition file
  -i [INTER], --inter [INTER]
                        interfield delimiter
  -j [INTRA], --intra [INTRA]
                        intrafield delimiter
  -u [USERNAME], --username [USERNAME]
                        username for API
  -p [PASSWORD], --password [PASSWORD]
                        password for API
  -q [QUERYURI], --queryuri [QUERYURI]
                        URI for API
  -r [RDFPREFIX], --rdfprefix [RDFPREFIX]
                        RDF prefix
  -x [URIPREFIX], --uriprefix [URIPREFIX]
                        URI prefix
  -s [SRC], --src [SRC]
                        name of source file containing data to be updated in
                        VIVO
  -c [CONFIG], --config [CONFIG]
                        name of file containing config data. Config data
                        overrides program defaults. Command line overrides
                        config file values
  -v, --verbose         write verbose processing messages to the log
  -n, --nofilters       turn off filters

For more info, see http://github.com/mconlon17/vivo-pump
```

See Getting Started for a description of `username` , `password` , `queryuri` , and `uriprefix` . Each is specific to your site and must be correct for Simple VIVO to work. We recommend putting these values in the Simple VIVO config file so that they do not have to be put on the command line.

Simple VIVO Reserved Column Names

Column names can be anything you like, but several column names are "reserved" -- Simple VIVO expects to use these names for its own work. If you try to use these column names, Simple VIVO will produce an error message and you will need to rename your column.

The table below lists the reserved column names in Simple VIVO.

Name	Use
uri	A column with the name `uri` is returned by every `get`. The column contains the Uniform Resource Identifiers of the things you working with in VIVO. Everything in VIVO has a URI.
action	If a column with the name `action` is included in a spreadsheet used in `update`, Simple VIVO will look in the action column for additional instructions regarding the processing of the row.

URIs in Updates

The `uri` column in your spreadsheet indicates to Simple VIVO how each row should be handled. There are several cases:

uri	Simple VIVO result
blank	The data on the row will be added. A uri will be created by Simple VIVO using the `uriprefix`. The data on the row will be added to VIVO referring to the new uri.
found uri	The data on the row will update the entity with the found uri.
not found uri	The data on the row will be added. The uri you specified, that was not found in VIVO, will be used.
invalid uri	If the value in the uri is not a valid uri, Simple VIVO will stop with an error message. In particular, the word "None" is not valid in the the uri column.

Additional Notes

1. The uri column is not like other data columns. blank means "create a uri". None is an invalid uri and will result in an error. A found uri indicates an update. An unfound uri indicates an add.
2. The way uris are used in update supports 'round tripping'. A get will return all the entities in VIVO along with their uri. A subsequent update using the uri found by get will result in updates to the uri in VIVO. And if you choose to add new rows with blank uri to the spreadsheet returned by a get, these rows will be assigned new uri. Everything works as expected.
3. The way uris are used in update supports loading data sets produced by others that contain their uri. If the uri in these datasets will not be generated by your `uriprefix`, then they can be added safely. if the dataset produced by others does not have uri, then they can be added safely -- Simple VIVO will generate uri using your `uriprefix`

A cautionary note

Perhaps you have noticed a potentially dangerous consequence of the uri processing rules. If your spreadsheet has URI in it that are URI of entities in your VIVO, **they will be treated as updates**. If you intended to add entities, you must make sure that the uri you specify will not *collide* with uri already in your VIVO.

Data sets created by others can have uri they create that will not be created by your `uriprefix` , or they can have no uri at all and Simple VIVO will create new ones that are not in your VIVO. But if a dataset created by others has uri in it, and those uri could be generated by your `uriprefix` , you could have a bad result, with the data produced by others being used to update existing VIVO entities rather than creating new ones.

Be sure that the uri used in your spreadsheet are not in your VIVO if you expect Simple VIVO to add them.

Merging Entities

On occasion, you may find that the same "thing" appears in VIVO twice -- the same person, the same organization, the same paper. This may happen when two different processes are being used to manage data -- automated processes are loading people while manual edits are also adding people. Or duplicates may occur because one or more of the entities is not completely identified. You may expect people to have ORCID identifiers, but perhaps a person has been added without one, and added again with one. Whatever the cause, you may find that when reviewing the rows in a spreadsheet, two or more of the rows are referring the same thing.

A merge is a way for you to combine two or more entities into a single entity. You will specify which entity you wish to keep (the primary entity) and which you wish to combine (the secondary entities) to the primary.

To specify a merge, you will need to designate a primary, and one or more secondaries to merge to the primary. If you do not specify a primary, no merge will occur. If you do not specify one or more secondaries, no merge will occur.

Specifying Primary and Secondary entities for Merge

To specify the primary and secondary entities for a merge, you will use the `action` column in your update spreadsheet. In the action column, use any text you like to specify the primary entity for the merge. You may choose to use the letter `a` for example. To specify the secondaries to merge to the primary, use the very same text followed by the digit `1`. So if you used `a` for the primary, you will use `a1` to indicate the secondary entities to merge to the primary.

You can specify as many primary entities for merging as you wish. And you may specify any number of secondaries to merge to the primary. And primaries and secondaries may appear in an any order in your spreadsheet.

An Example of Merging Organizations

Suppose your organizational data looks like:

uri	name
http://vivo.yourschool.edu/n8828	Harvard University
http://vivo.yourschool.edu/n8910	University of Iowa
http://vivo.yourschool.edu/n1402	Harvard University
http://vivo.yourschool.edu/n2245	Stanford University
http://vivo.yourschool.edu/n2249	Cambridge University
http://vivo.yourschool.edu/n8991	Harvard University
http://vivo.yourschool.edu/n9012	University of Toronto
http://vivo.yourschool.edu/n9000	Cambridge University
http://vivo.yourschool.edu/n9234	Bucknell University
http://vivo.yourschool.edu/n9231	Louisiana State University
http://vivo.yourschool.edu/n9432	Harvard University

We see that there are several copies of Harvard and two copies of Cambridge. Upon inspection we determine that `http://vivo.yourschool.edu/n1402` should be the primary and all others secondary for a merge of the Harvard entities and that `http://vivo.yourschool.edu/n9000` should be the primary for Cambridge and the other the secondary.

Your update spreadsheet will look like the one below. We use `a` to indicate the entities involved in the Harvard merge, and `b` to indicate the Cambridge entity merge participants. There are four Harvard rows. One is the primary the other three are secondary. There are two Cambirgde rows. One is the primary and the other is secondary.

All other rows have a blank in the action column. As always in Simple VIVO, blank means "do nothing." These rows will not be involved in the merge.

uri	name	action
http://vivo.yourschool.edu/n8828	Harvard University	a1
http://vivo.yourschool.edu/n8910	University of Iowa	
http://vivo.yourschool.edu/n1402	Harvard University	a
http://vivo.yourschool.edu/n2245	Stanford University	
http://vivo.yourschool.edu/n2249	Cambridge University	b1
http://vivo.yourschool.edu/n8991	Harvard University	a1
http://vivo.yourschool.edu/n9012	University of Toronto	
http://vivo.yourschool.edu/n9000	Cambridge University	b
http://vivo.yourschool.edu/n9234	Bucknell University	
http://vivo.yourschool.edu/n9231	Louisiana State University	
http://vivo.yourschool.edu/n9432	Harvard University	a1

You use the spreadsheet with the action column to update VIVO as always:

```
python sv.py -a update
```

You add and remove RDF from VIVO. Use a get to see the result:

```
python sv.py -a get
```

Your spreadsheet will looks like the one below:

uri	name
http://vivo.yourschool.edu/n8910	University of Iowa
http://vivo.yourschool.edu/n1402	Harvard University
http://vivo.yourschool.edu/n2245	Stanford University
http://vivo.yourschool.edu/n9012	University of Toronto
http://vivo.yourschool.edu/n9000	Cambridge University
http://vivo.yourschool.edu/n9234	Bucknell University
http://vivo.yourschool.edu/n9231	Louisiana State University

The duplicate Harvard secondary entities have been merged to the primary Harvard entity and are no longer found in VIVO. Anything associated with those entities is now associated with the primary (and only) Harvard entity. The same has happened for the secondary Cambridge entity.

The Difference Between Merge and Remove

We might have used `Remove` in the `action` column to remove all the secondary entities and leave only the primaries.
What is the difference remove and merge?

The difference has to do with things associated with the secondary entities. Organizations might have addresses, or phone numbers, web pages, sub organizations or other data. In a merge, all the data from the secondaries is associated the primary. Nothing is lost. In a remove, all the data from the secondaries is removed as well. Associations and data values associated with the secondary are removed.

Both operations may have negative consequences:

1. In a remove, data may be lost.
2. In a merge, multiple values for the same attribute may be associated with the primary. You may need to "clean up" the primary after a merge to insure it has the data values you want, and only the data values you want.

In general, use `remove` only in very simple cases where the entity to be removed has *no* data of interest.

Use `merge` to combine partial copies of entities. But always be prepared to clean up the resulting primary entity.

Managing More Data

The examples provided with Simple VIVO manage quite a bit of data, and many of the most common scenarios for keeping track of the scholarly work of the people of the institution. But your institution may wish to extend the scholarly record to track more data. You may wish to track datasets and/or software created by faculty, or patents obtained, or resources used, or projects completed. There is no end to the types of work you can track with Simple VIVO.

In this chapter we will look at extending the delivered scenarios by adding attributes to the existing scenarios, by adding new scenarios for scholarly work already modeled in VIVO, and finally, by adding completely new types of work.

Refining the Capture of the Scholarly Record

Capturing More of the Scholarly Record

Extending the Nature of the Scholarly Record

The Simple VIVO Configuration File

The Simple VIVO configuration (config) file tells Simple VIVO about your VIVO, what you would like to have done, and how you would like to do it. Simple VIVO command line parameters override the values in the config file. In this way, the config file can say what you usually do, and the command line can say what you would like done in this particular run.

For example, if you have a config file that tells Simple VIVO everything it needs to know to update the people in your VIVO, you can run Simple VIVO using

```
python sv.py
```

No parameters are needed. But if you would like to see all the internal messages that Simple VIVO generates while it is working, you can use:

```
python sv.py -v
```

You can get a complete list of all the command line parameters by using

```
python sv.py -h
```

By default, Simple VIVO expects your config file to be named `sv.cfg`. But you can call it whatever you would like and tell Simple VIVO the name of your config file on the command line.

For example, suppose you name your config file 'uf_production_sv.cfg`, which might be a good name if you are at the University of Florid and your config file is describing the production environment. You would run Simple VIVO using:

```
python sv.py -c uf_production_sv.cfg
```

Config file parameters

A sample config file `sv.cfg` is shown below.

Sections

Sections have names of the form `[sectionname]` There are three section headers in the config file. They can appear in any order.

`[sparql]` specifies parameters to be used to access your VIVO.

`[pump]` specifies parameters to be used to perform Pump operations.

```
[sparql]
queryuri = http://localhost:8080/vivo/api/sparqlQuery
username = vivo_root@school.edu
password = v;bisons
prefix =
    PREFIX rdf:   <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
    PREFIX rdfs:  <http://www.w3.org/2000/01/rdf-schema#>
    PREFIX xsd:   <http://www.w3.org/2001/XMLSchema#>
    PREFIX owl:   <http://www.w3.org/2002/07/owl#>
    PREFIX vitro: <http://vitro.mannlib.cornell.edu/ns/vitro/0.7#>
    PREFIX bibo: <http://purl.org/ontology/bibo/>
    PREFIX event: <http://purl.org/NET/c4dm/event.owl#>
    PREFIX foaf: <http://xmlns.com/foaf/0.1/>
    PREFIX obo: <http://purl.obolibrary.org/obo/>
    PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
    PREFIX uf: <http://vivo.school.edu/ontology/uf-extension#>
    PREFIX vitrop: <http://vitro.mannlib.cornell.edu/ns/vitro/public#>
    PREFIX vivo: <http://vivoweb.org/ontology/core#>
[pump]
action = get
verbose = false
inter = tab
intra = ;
nofilters = false
defn = position_def.json
src = position_update_data.txt
uriprefix = http://vivo.school.edu/individual/n
```

Parameters

Parameters take the form `name = value`. All parameters are optional. Parameters must appear in the appropriate section. Within the section, the parameters may appear in any order. The parameters that may appear in each section are defined below.

Section	Possible parameters
sparql	queryuri, username, password, prefix
pump	action, verbose, inter, intra, nofilters, defn, src, uriprefix

Each is described below.

`queryuri` gives the URI of the SPARQL API interface to your VIVO. That is, the URI that will be used to make queries to your VIVO. This URI is new in VIVO 1.6. The Pump requires VIVO 1.6 or later. The sample value is for a VIVO Vagrant, a test instance.

`username` gives the username (VIVO may call this "email") for an account that is authorized on your VIVO to execute the VIVO SPARQL API.

`password` gives the password of the account that is authorized on your VIVO to execute the VIVO SPARQL API. That is, the password to the account specified by the `username`

`prefix` gives the SPARQL prefix for your VIVO. SPARQL queries typically use PREFIX statements to simplify the specification of URI in SPARQL statements. The value of the prefix parameter should be the collection of PREFIX statements used at your VIVO. Note that formatting of the parameter -- each PREFIX statement begins on a new line, including the first one. Each is indented by four spaces. This is recommended.

Note: All parameters are optional. If a parameter is not specified, the Pump software will supply a default value. But the default values for the Pump software are set for a VIVO Vagrant, not your VIVO. You should consider the four parameters above to be required for your `sv.cfg`.

`action` is the desired Pump action. There are four choices: update, get, summarize and serialize. Often these are specified on the command line to override the value in the config file and make explicit what action Simple VIVO is performing.

`verbose` is set to `true` to have the Pump produce output regarding the status and progress of its actions. The default is `false` which produces limited output showing the the datetime the action started, the number of entities effected, and the datetime the action was completed.

`inter` is the inter field separator character used for your spreadsheets. a CSV file would have `inter = ,` , a tab separated spreadsheet (recommended) would have `inter = tab`. Note the word "tab". the Simple VIVO config file uses this to specify a tab as a field separator. Another popular choice for field separator is the "pipe" character "|". Pipe and tab are used as separators because these characters do not appear in literal values in VIVO.

`intra` is the intra field separator character used to separate multiple values in a single spreadsheet cell. The default is a semicolumn.

`nofilters` set to `true` indicates that filters specified in the definition file should not be used. Set to `false` indicates filters are to be used as normal.

`defn` specifies the name of the file that contains the definition file, in JSON format, to be used by Simple VIVO.

`src` specifies the name of the input source file (for update) or output file (for get).

`uriprefix` indicates the format of any URI to be created by the update process. URI will have random digits following the prefix. So, for example, if your prefix is `http://vivo.school.edu/n`, Simple VIVO will create new entities with URI that look like `http://vivo.school.edu/nxxxxxxxx` where `xxxxxxxx` are random digits.

Simple VIVO Test Cases

Simple VIVO can be tested using `test_sv.py` in the `tests` directory. Various tests are performed and test results reported along with timings of each test.

Running the Tests

To run the tests, open a terminal window, navigate to the `tests` directory and use:

```
python test_sv.py
```

Test results will be displayed in your terminal window. Sample test output is shown below. Three items are displayed for each test:

1. The name of the test. `config not found` is the name of a test that demonstrates Simple VIVO behaviour when the configuration file is not found.
2. The return code. Simple VIVO returns `0` if no errors occured. Simple VIVO returns `1` if errors occurred. For the `config not found` test, the configuration file was not found. This is an error. Simple VIVO produces a return code of `1` .
3. The time in seconds to complete the test. Timings were done with a Simple VIVO and a VIVO Vagrant running on a MacBook.

```
         config not found     1     0.112974
definition file not found     1     0.044921
                     help     0     0.052848
    source file not found     1     0.114163
                     test     0     0.880568
```

Introduction to the Pump

The VIVO Pump (under development), provides a means for managing VIVO data using spreadsheets. Rows and columns in spreadsheets represent data in VIVO. The Pump update method takes spreadsheet data, and a definition file, and returns VIVO RDF (add and subtract) for updating VIVO with the values in the spreadsheet. The Pump get method uses the definition file and the data in VIVO to create a spreadsheet. Schematically:

```
update method:  Spreadsheet Data    =>   The Pump      =>    VIVO RDF   =>   VIVO
                                       + definition file

get method:     Spreadsheet Data    <=   The Pump      <=    VIVO RDF   <=   VIVO
                                       + definition file
```

The same definition file can be used for get and update, and the spreadsheet resulting from a get can be used in a subsequent update, providing a "round trip" capability. That is, you can "get" data, improve it manually or programmatically, and use it to "update" VIVO.

Work in Progress

The Pump is under active development, and beta testing at the University of Florida.

See Issues for work in progress, features to be added, and bugs to be addressed.

Testing the Pump

- Pump Test Cases
- Simple VIVO Tests

Pump References

- The Pump API
- The Pump Definition File

Using the Pump

The Pump can be used from the command line, using a delivered main program called Simple VIVO (`sv`)

```
python sv.py -defn my_definition_file.json -src my_spreadsheet_file.csv -a get
```

By default, Simple VIVO reads parameters from a configuration file. The configuration file tells Simple VIVO the URL of your VIVO SPARQL API, and the username and password of an account that will be used to perform updates.

The Pump can also be used from a Python program by importing the pump, creating a pump object and calling a method on it. In the example below, a Pump object is created with a definition file and a source file. The `get()` method uses the definition file to build queries that are run in your VIVO and used to produce the source file.

```
import pump
p = Pump(defn='my_def.json', src='my_src.txt')
p.get()
```

In the example below, a Pump is created with a definition file name and a source file name. An update is performed. The update uses the definition to identify data in VIVO and compares it to data in the source file. Source file data values are used to update VIVO. Following the update, VIVO contains the values from the source file.

```
import pump
p = Pump(defn='my_def.json', src='my_src.txt')
p.update()
```

Writing Definitions

The Pump uses a definition file to define how data can be retrieved from VIVO to a spreadsheet, and how data in a spreadsheet can be updated in VIVO. A single definition file defines both the get action and the update action.

Round tripping is the ability to get data from VIVO to a spreadsheet, edit that spreadsheet and use the same definition to update VIVO. You can also start from the update -- adding new data to VIVO, and then using the same definition to get the data from VIVO.

In this chapter we will see simple examples first. You should try these simple examples. And you should check your work. See The Pump Definition File Reference for all definition file features.

A simple definition file

Definition files are written using JSON syntax. JSON is a popular file format, but can be difficult to write by hand. You should have a text editor that understands JSON format. The text editor will help you make sure that the definitions you write are correct JSON syntax.

Rows and Columns

To write a definition file, you need to think about the spreadsheet you will use to manage your VIVO data. What do the rows in your spreadsheet represent? Do you have one row per person? Per publication? Per academic degree? VIVO is capable of storing many kinds of data -- actually, VIVO is capable of managing all kinds of data. VIVO manages whatever its ontologies define. The rows in your spreadsheet may represent any of the kinds of things that your ontologies define.

Second, you will need to consider what data are stored in the columns of your spreadsheet. Each column represents an attribute of the entity represented by the row.

Suppose you are managing data about people. For each person, you have their name, and their academic department.

name	department
Conlon, Michael	Statistics
Barnes, Christopher	Informatics

The rows of your spreadsheet represent people. The first column represents the name of the person. The second column represents the department[1] of the person.

Writing an entity_def

Writing a simple column_def

Literals

Objects and Enumerations

Writing a two step column_def

Writing a three step column_def

Writing a closure_def

More features of definition files

[1]. In universities, identifying the department of the person may not be straightforward. There is the concept of "tenure department," and "appointment department," and "department paying the majority of salary," and "department where the person resides. Many faculty have appointments in multiple departments. For the purpose of the example, we will consider department "knowable." ↩

The Pump Definition File Reference

Every use of the Pump is defined by a pump definition file, which describes the correspondence between the data in the spreadsheet and the data in VIVO. Rows in the spreadsheet correspond to entities in VIVO. The rows in a spreadsheet might represent people, for example. Columns in the spreadsheet represent "paths" to attributes in VIVO. This will become clearer as we see examples of how paths are used to define the way in which the Pump will get or update values in VIVO. The cells in the spreadsheet represent attribute values in VIVO.

The pump definition file is always in JSON format.

Minimum definition

A minimum definition includes the required elements (marked with an asterisk below) in the required structure.

The definition file shown below is for a simple example maintaining cities (VIVO calls these PopulatedPlaces) each with two columns -- a name and a US state. For example, Chicago is in Illinois.

The entity_def defines the rows of the spreadsheet. The column_defs define the columns of the spreadsheet. Each column def is a list of "paths." Each path is a dictionary. Each dictionary has two named elements -- "predicate" and "object". The predicate element is also a dictionary, as is the object element. This is the required structure. In some cases, a definition will require "closures" to define additional relationships between columns.

The remainder of this document describes the elements, their purpose and whether they are required.

```
{
    "entity_def": {
        "entity_sparql": "?uri a vivo:PopulatedPlace .",
        "type": "vivo:PopulatedPlace"
    },
    "column_defs": {
        "name": [
            {
                "predicate": {
                    "single": true,
                    "ref": "rdfs:label"
                },
                "object": {
                    "literal": true
                }
            }
        ],
        "state": [
            {
                "predicate": {
                    "single": true,
                    "ref": "obo:BFO_0000050"
                },
                "object": {
                    "literal": false,
                    "enum": "states_enum.txt"
                }
            }
        ]
    }
}
```

entity_def*

The entity_def includes elements used to describe the rows of the spreadsheet.

entity_sparql*

Entity sparql will be used by the Pump to select the entities to represent the rows in your spreadsheet. You may have a simple entity_sparql such as:

```
?uri a foaf:Person .
```

The rows in your spreadsheet will be all people in your VIVO.

But you may need further restriction, as in:

```
?uri a vivo:FacultyMember . ?uri ufVivo:homeDept <http://vivo.ufl.edu/ndividual/n12312
3> .
```

where we are asking for all faculty members in the Chemistry department. Using a restricted `entity_def` , you can limit the number of rows in your spreadsheet, or partition the data management task into pieces to be done by different people.

`entity_sparql` always refers to the entities to be select as " `?uri` . `entity_sparql` must be valid SPARQL statements, each ending with a period as shown above.

type*

Indicates the rdf:type of each row of the spreadsheet. In an update, each new entity will have this type. The VIVO inferencer will supply "parent" types from the ontology. So, for example, indicating that the rows are type PopulatedPlace, the inferencer will supply the parents: Continuant (obo), Entity (obo), Geographic Location (vivo), Geographic Region (vivo), Geopolitical Entity (vivo), Immaterial Entity (obo), Independent Continuant (obo), Location (vivo), Populated Place (vivo), Spatial Region (obo), Thing.

order_by

Indicates the name of a column or columns that will be used to order the resulting spreadsheet. If no order by is given, the resulting data will be provided in URI order.

column_defs*

The column_defs specify the columns of the spreadsheet. There is one column name for each column in the spreadsheet.

Two column names are reserved words, used by the Pump:

Name	Purpose
uri	will contain the URI of the entity.
remove	The remove column is optional.

If present, it can be used to remove entities from VIVO during an update by putting the word "remove" on a row in the spreadsheet in the remove column.

For a get, the column_defs define the output spreadsheet. The resulting spreadsheet will contain one column for each column_def, plus a URI column. The columns in the spreadsheet will appear in the order of the column_defs in the definition file.

For an update, the the source spreadsheet may contain columns not defined in the definition file. These columns are ignored. The definition file may contain column_defs not found in the source file. These column_defs are ignored. RDF is generated only for columns found in *both* the column_defs and the columns of the source spreadsheet.

Column name*

Column names are specified in the definition file as shown in the example ("name" and "state"). Column names can be any length, upper and or lower case, with or without punctuation and spaces.

Whatever appears as a column name in the definition file will appear as a column name in the output spreadsheet produced by get.

In an update, the column name in the source must match the column name in the definition file *exactly*. Simple column names are preferred as a consequence.

The column_def is a list*

Each column_def is a list, as indicated by the brackets [ ]. These lists describe the "path" from the entity to the attribute represented by the column value. Paths may be of length 0, 1, 2 or 3. Lists of length zero are used to specify desired columns that will not be processed. For example:

```
notes: [
]
```

is valid and indicates that the spreadsheet has (update) or will have (get) a column called "notes". Nothing will be done with this column.

A length one list indicates that the entity has a predicate which has an object. Many VIVO attributes can be specified using length one paths. A common example is label. The label of an entity is specified using a triple of the form

```
<entity_uri>  rdfs:label  "label_literal_value"
```

See the cities example above. It specified that the first column of the spreasheet will be called "name". This column is then defined to be a length 1 path (the list has one element). The element is a dictionary as indicated by the brackets. The dictionary has two key values, "predicate" and "object". This is *required*. Every column definition is a list, every list element is a dictionary, every dictionary has two key values, "predicate" and object".

So for a length one path, this looks like:

```
"name": [
    {
        "predicate": {
        },
        "object": {
        }
    }
]
```

So for a length two path, this looks like:

```
"name": [
    {
        "predicate": {
        },
        "object": {
        }
    },
    {
        "predicate": {
        },
        "object": {
        }
    }
]
```

There are two dictionaries in the list. Each dictionary has exactly two elements. One element must have the key "predicate," the other has the key "object." The length two path definition indicates that the entity has a predicate that refers to an intermediary object, and that object in turn has a predicate and that the object of the second predicate is the attribute that appears in the spreadsheet. While this appears to be complex at first, it becomes clearer as examples are studied, and figures showing these relationships are examined.

Length three paths are similar. The length three column_def looks like:

```
"name": [
    {
        "predicate": {
        },
        "object": {
        }
    },
    {
        "predicate": {
        },
        "object": {
        }
    },
    {
        "predicate": {
        },
        "object": {
        }
    }
]
```

And the meaning is analogous to the length two path. The entity has a predicate the refers to an intermediate object that in turn has a predicate referring to a second intermediate object that in turns refers to an object that is the attribute that appears in the spreadsheet. VIVO has a number of three length paths attributes as you will see in the examples.

There are no four length paths in VIVO and length four paths are not supported by the Pump. Definition files with length four paths will be flagged as invalid.

predicate*

The "predicate" key name is required in each element of the path. See above. By setting values of the predicate (see below) you can define the path from the entity to the value in the spreadsheet.

ref*

Used to specify the uri of the predicate.

Example: You wish to manage people. You have a column in the spreadsheet called "name." You want to associate values in that column to the `rdfs:label` of the person entity. To do this, you set the "ref" attribute of the "predicate" entry to "rdf:label" You may use all the prefixes that you are supplying to the Pump through its `prefix` argument.

single*

Used to indicate whether the predicate is single valued (true), multi-valued (false), or boolean.

If you specify "true" for single, the Pump will warn you if it finds multi-valued values for the predicate. Your value will replace *all* the values in VIVO.

If you specify "false" for single, the Pump will expect you to supply all the values for the predicate. If you specify one value, *all* the values in VIVO will be replaced with the value you specify. If you specify 3 values and VIVO has two, the Pump will compare the values you specify with the values in VIVO and add and remove values as needed to insure that the values you specify are the values with that will be in VIVO.

If you specify "boolean" for single, the Pump will process the column according to the value property in the object.
For a `get`, if the value is found, the column will contain a '1', otherwise a '0'. For an `update`, a '1' indicates that the value should be added if not present. A '0' indicates the value should be removed if found. 'None' can be used as always to remove the value. Blank, as always, indicates do nothing.

Example: VIVO has values "a" and "b" for an attribute. You specify the attribute is "single": false. You specify values "b";"c";"d" VIVO will remove "a", leave "b" as is, and add "c" and "d" as values for the attribute.

A common case for multiple values is type of an entity. Many entities have multiple types. An organization may be a funding organization and a research organization, for example. When specifying "single": false, you must specify all the values.

"boolean" is used to indicate that the column is a simple indicator of whether the predicate and the object value should be added or removed from VIVO. If '' appears in the column, no action is taken. If '0' or 'None' or 'n' or 'no' or 'false' appears in a boolean column, the object value is removed from VIVO if present. If any other value is found in the column, the object value is added to VIVO if not currently in VIVO. Whenever "boolean" is specified, the "object" (see below) must have a "value" (see below). Failure to provide a value for an object with a boolean predicate is considered an invalid definition.

include

Used to specify values that must be included by default in a multi-valued predicate.

include is optional. Use include when you want to be sure that the multi-valued predicate always includes the values you give in include. include is a list. You can specify as many values as you need in an include list.

Not: You do not need to include inferred values in an include list. The VIVO inferencer will supply additional types for entities.

object*

The "object" key name is required in each element of the path. See above. The "object" key name may contain one or more of the parameters described below.

datatype

Specifies the datatype of the literal object. xsd datatypes are supported:

datatype	Usage
xsd:integer	When literal values must be integers. Ranks, for example.
xsd:string	When literal values are strings.
xsd:datetime	For all VIVO dates and date times
xsd:boolean	Rare
xsd:decimal	For decimal numbers suh as dollar amounts
xsd:anyURI	For literal values that are URI, wuch as web page addresses

literal*

True if the object is a literal value, False if the object is a reference to an entity.

enum

Used to specify enumerations that are used to substitute one value for another. Enumerations are very handy in VIVO data management.

filter

Filters are simple functions that take the value supplied and "improve" it, typically by correcting misspellings, formatting, expanding abbreviations, improving the use of upper and lower case, and other simple improvements.

The Pump provides filters which can be used to improve the data quality of VIVO.

Filter	Usage
improve_title	Used on publication and grant titles
improve_course_title	SPEC TOP IN AGRIC => Special Topics in Agriculture
improve_jobcode_description	AST PROF => Assistant Professor
improve_email	Returns RFC standard email addresses
improve_phone_number	Returns ITU standard phone numbers
improve_date	Returns yyyy-mm-dd
improve_dollar_amount	Strings which should contain dollar amounts (two decimal digits)
improve_sponsor_award_id	For NIH style sponsor award ids
improve_deptid	For department identifiers (eight digits, may start with zero)
improve_display_name	When displaying the whole name (not name parts) of a person
improve_org_name	Organization labels

Example for publication titles:

```
"title": [
    {
        "predicate": {
            "single": true,
            "ref": "rdf:label"
        },
        "object": {
            "filter": "improve_title",
            "literal": true
        }
    }
]
```

handler (future)

A handler can be added to an object to indicate that special processing is required in addition to the normal operation of the pump. An example is photo processing. The name of the photo file is the object literal value, but this is not sufficient for a photo to appear in VIVO. The photo file must be copied to a specific location on the VIVO server. A photo handler can perform this action.

Note: Handlers are a future feature and are not available in the current version of the Pump.

type

Intermediate objects in multi-length paths will be generated as needed by an update. These new intermediate objects will be given the type as specified. The VIVO inferencer will supply all parent types.

value

value is used when the predicate processing is boolean. A value for the object is specified using the value parameter.
If the column contains a '0' or 'None' or 'false' or 'n' or 'no', the value is removed from VIVO if present.
If any other value is found in the column, the value is added to VIVO if not currently there.
Blank column values are not processed.

Example: A new research area 'x85' is created and 50 faculty are identified that have the research area. A spreadsheet of faculty members and an x85 column is defined as boolean with the value 'http://definitionsource/x85'.
The spreadsheet contains a '1' in the x85 column for each of the faculty members with the x85 research area.
All other data for the faculty members is unchanged. The pump will add hasResearchArea assertions for each of the fifty faculty. The definition is shown below:

```
"x85": [
    {
        "predicate": {
            "single": "boolean",
            "ref": "vivo:hasResearchArea"
        },
        "object": {
            "literal": false,
            "value": "http://definitionsource/x85"
        }
    }
],
```

lang

The RDF Lang attribute can be specified for the object. An example is shown below

```
"name": [
    {
        "predicate": {
            "single": true,
            "ref": "http://www.w3.org/2000/01/rdf-schema#label"
        },
        "object": {
            "literal": true,
            "lang": "en-US"
        }
    }
],
```

Object literal values will have the lang attribute in the RDF.

qualifier

Use a qualifier to specify an additional condition which must be met to specify the path to the attribute.

label

Use label to specify a label to be added to the intermediate object constructed by the Pump during an update.

closure_defs

Closures are used to define additional paths required to specify the semantic relationships between between entities.
In most cases, closures are not needed -- paths from the row entity can be defined through the column_defs from row entity to the column value, whether literal or reference. But in some cases, there are relationships between entities referred to from the row entity.

Closure definitions are identical in all ways to column_defs. They are processed after all column_defs regardless of where they appear in the definition file.

For example, in teaching, the row entity, TeacherRole, has a clear path to the teacher and to the course. But there is also a relationship to be defined between the person and the course. One might argue that this relationship is unnecessary and that would be correct. The relationship could be inferred. But our semantic inference is not yet capable of making the inference between person and course, given the relationships between TeacherRole and person, and TeacherRole and course. A closure is to to make the assertion between Person and Course.

Just as with any normal column_def, the closure defines a path from the row entity to the column value. In the teaching example, we could define a closure from the TeachingRole to the Course through the Person, thereby defining the relationship between the Course and the Person. Or we could define the closure a length 2 path from TeacherRole to Person through the Course, thereby defining the relationship between the Person and the Course. The two paths produce the same result -- the Person participates in the Course and the Course has a participant of the Person. The Closure defines this relationship.

The syntax of the closure_def is identical to the syntax of the column_def. Any number of closures can appear and in any order. All elements of the column_def are available in closure_defs and work the same way.

The closure_def is just a second chance to use the column data to define additional paths required to represent the relationships between the entities.

For the teaching example, the closure defines a second path from the TeachingRole (row entity) to the course (column entity) through the instructor. The `closure_def` is shown below:

```
"closure_defs": {
    "course": [
        {
            "predicate": {
                "single": true,
                "ref": "obo:RO_0000052"
            },
            "object": {
                "literal": false,
                "type": "foaf:Person",
            }
        },
        {
            "predicate": {
                "single": true,
                "ref": "obo:BFO_0000056"
            },
            "object": {
                "literal": false,
                "enum": "data/course_enum.txt"
            }
        }
    ]
},
```

Notes

1. The course is "reused" by the closure to define a second path from the TeacherRole to

the course. See the column_def for the first path.

2. The first step in the path is through the instructor.
3. The instructor step in the path does not need a qualifier. It is coming from the row entity and there is only one entity that has the predicate relationship to the row entity.

The Pump API

The Pump API is the collection of exposed elements that define how other software and users interact with the Pump.

The Pump API consists of:

1. The Pump arguments. All the things that are set by default and/or are available to the programmer and the user to set to define the actions of the Pump. All arguments are accessible via command line arguments of the main program `sv` (Simple VIVO) which is used to call the Pump.
2. The Pump methods and instance variables. Pump methods and instance variables are accessible to the programmer and so constitute part of the Pump API.
3. The Pump Definition file. Each Pump requires a definition file, in JSON format, to define the transformations the Pump will perform to and from the flat file representation to the semantic web representation.

Elements of the distribution that do not constitute parts of the Pump API, include **everything else**, including, but not limited to:

1. Data
2. Examples
3. The `vivopump` library, which consists of functions used by the Pump.
4. Test cases
5. Config files
6. Documentation files

Definition of the Pump API is critical for understanding Semantic Versioning used in the Pump.

Pump Arguments

The Pump has two arguments. Each has a default. They can appear in any order.

`defn` gives the name of the JSON definition file. The default is `pump.json`

`src` gives the name of the file containing the CSV data to be used for an update, or, for a get action, the name of the file that will be created by the pump. The default is `pump.txt`

Pump instance variables

Any of the following can be set to further define the work of the Pump.

`queryuri` gives the URI of the SPARQL API interface to your VIVO. That is, the URI that will be used to make queries to your VIVO. This URI is new in VIVO 1.6. The Pump requires VIVO 1.6 or later. The sample value is for a VIVO Vagrant, a test instance.

`username` gives the username (VIVO may call this "email") for an account that is authorized on your VIVO to execute the VIVO SPARQL API.

`password` gives the password of the account that is authorized on your VIVO to execute the VIVO SPARQL API. That is, the password to the account specified by the `username`

`prefix` gives the SPARQL prefix for your VIVO. SPARQL queries typically use PREFIX statements to simplify the specification of URI in SPARQL statements. The value of the prefix parameter should be the collection of PREFIX statements used at your VIVO.

`action` is the desired Pump action. There are five choices: test, update, get, summarize and serialize. When the pump is used by Simple VIVO, the action is typically specified on the command line to override the value in the config file and make explicit what action Simple VIVO is performing.

`inter` is the inter field separator character used for your spreadsheets. a CSV file would have `inter = ,` , a tab separated spreadsheet (recommended) would have `inter = tab` . Note the word "tab". the Simple VIVO config file uses this to specify a tab as a field separator. Another popular choice for field separator is the "pipe" character "|". Pipe and tab are used as separators because these characters do not appear in literal values in VIVO.

`intra` is the intra field separator character used to separate multiple values in a single spreadsheet cell.
The default is a semi-column.

`filter` (default `true`) set to `true` indicates that filters specified in the definition file should be used. Set to `false` indicates filters are not to be used.

`rdfprefix` indicates the naming convention to be used by the pump when creating RDF files. The Pump will create add and sub RDF with the names `rdfprefix`_add.rdf and `rdfprefix`_sub.rdf

`uriprefix` indicates the format of any URI to be created by the update process. URI will have random digits following the prefix. So, for example, if your uriprefix is `http://vivo.school.edu/n`, Simple VIVO will create new entities with URI that look like `http://vivo.school.edu/nxxxxxxxx` where `xxxxxxxx` are random digits.

Pump Methods

The methods and instance variables of the Pump are used by programmers to define and perform the work of the pump.

Pump Methods

The Pump has four methods. None of the pump methods have parameters. Each is called simply by name. See below.

summarize

The summarize() method describes the operation that will be performed by the Pump as configured. It lists the values of the instance variables, the enumerations, the get_query, and the update_def. A single string with newlines is produced.

```
p = Pump()
print p.summarize()
```

serialize

The serialize() method produces a string version of the instance variables and update definition.

```
p = Pump()
print p.serialize()
```

Future: Should be suitable for round tripping back into the Pump as a state.

test

The test() method uses the current configuration to test the connection to your VIVO.

```
p = Pump()
print p.test()
```

A sample output is shown below:

```
2015-11-05 09:17:06.206090 Test results
Update definition    pump_def.json read.
Source file name    pump_data.txt.
Enumerations read.
Filters    True
Intra field separator    ;
Inter field separator
VIVO SPARQL API URI    http://localhost:8080/vivo/api/sparqlQuery
VIVO SPARQL API username    vivo_root@school.edu
VIVO SPARQL API password    v;bisons
VIVO SPARQL API prefix
PREFIX rdf:      <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs:     <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd:      <http://www.w3.org/2001/XMLSchema#>
PREFIX owl:      <http://www.w3.org/2002/07/owl#>
PREFIX swrl:     <http://www.w3.org/2003/11/swrl#>
PREFIX swrlb:    <http://www.w3.org/2003/11/swrlb#>
PREFIX vitro:    <http://vitro.mannlib.cornell.edu/ns/vitro/0.7#>
PREFIX wgs84:    <http://www.w3.org/2003/01/geo/wgs84_pos#>
PREFIX bibo:     <http://purl.org/ontology/bibo/>
PREFIX c4o:      <http://purl.org/spar/c4o/>
PREFIX cito:     <http://purl.org/spar/cito/>
PREFIX event:    <http://purl.org/NET/c4dm/event.owl#>
PREFIX fabio:    <http://purl.org/spar/fabio/>
PREFIX foaf:     <http://xmlns.com/foaf/0.1/>
PREFIX geo:      <http://aims.fao.org/aos/geopolitical.owl#>
PREFIX obo:      <http://purl.obolibrary.org/obo/>
PREFIX ocrer:    <http://purl.org/net/OCRe/research.owl#>
PREFIX ocresd:   <http://purl.org/net/OCRe/study_design.owl#>
PREFIX skos:     <http://www.w3.org/2004/02/skos/core#>
PREFIX uf:       <http://vivo.school.edu/ontology/uf-extension#>
PREFIX vcard:    <http://www.w3.org/2006/vcard/ns#>
PREFIX vitro-public: <http://vitro.mannlib.cornell.edu/ns/vitro/public#>
PREFIX vivo:     <http://vivoweb.org/ontology/core#>
PREFIX scires:   <http://vivoweb.org/ontology/scientific-research#>
Prefix for RDF file names    pump
Uriprefix for new uri    http://vivo.school.edu/individual/n
Sample new uri    http://vivo.school.edu/individual/n3171760016
Simple VIVO is ready for use.
2015-11-05 09:17:06.812863 Test end
2015-11-05 09:17:06.812952 Finish
```

get

get() performs the Pump get operation, querying VIVO according to the definition, creating a spreadsheet data structure, and writing that structure to the output file specified by the Pump instance variables. get() returns the number of rows in the output spreadsheet.

~~~
p = Pump()
nrows = p.get()

### update

update() performs the Pump update operation, using source data provided in a spreadshe
et and a definition file supplied as JSON.  The update queries VIVO for current values
, adds and removes entities as directed in
the spreadsheet and adds, removes and updates attributes values as supplied in the spr
eadsheet.  update() returns an add graph and a subtract graph, which can be serialized
 using rdflib.  For example:

```python
[add_graph, sub_graph] = p.update()
add_file = open(args.rdfprefix + '_add.rdf', 'w')
print >>add_file, add_graph.serialize(format='nt')
add_file.close()
~~~

Pump Test Cases

The VIVO Pump is deceptively simple. A spreadsheet is used to update elements in VIVO. What could go wrong? Here we provide an overview of tests cases of various kinds.

Entity Level Tests

1. No-op. Round trip the entities by doing a get and an an update from the resulting spreadsheet. Nothing should be added or subtracted. NOTE: If filters are used, the get will "improve" the data and the first round trip will make filter improvements to the data. Subsequent round trips will be operating on the improved data and should then be a no-op.
2. Add entity. Create new entities as per the definition.
3. Change entities. See below. Changes are at the element level.
4. Delete entities. This is done using the "remove" column in the spreadsheet.

Element level Tests

Individual elements in the spreadsheet are identified by row (entity) and column (path definition). Path definitions lead to objects in VIVO triples. Tests are conducted with and without qualifiers.

Unique	Length	ADC	Qual	Examples
Yes	1	Add	No	Building Abbreviation
Yes	1	Change	No	Building Abbreviation
Yes	1	Delete	No	Building Abbreviation
Yes	2	Add	No	Grant start date
Yes	2	Change	No	Grant start date
Yes	2	Delete	No	Grant start date
Yes	3	Add	No	Org zip code
Yes	3	Change	No	Org zip code
Yes	3	Delete	No	Org zip code
No	1	Add	No	Person type(s); Person Research Area(s)
No	1	Change	No	Person type(s); Person Research Area(s)
No	1	Delete	No	Person type(s); Person Research Area(s)
No	2	Add	No	Grant Investigators
No	2	Change	No	Grant Investigators
No	2	Delete	No	Grant Investigators
No	3	Add	No	No example
No	3	Change	No	No example
No	3	Delete	No	No example

Functional and component tests

The definitions support various functions through components of the software. Examples include enumeration and filtering. Each must be tested.

1. Component tests. Each software component needs unit tests.
2. Filter tests. Unit tests of filters. Round trip testing of filters.
3. Enumeration tests. Unit tests of enumerations. Round trip testing of enumerations.
4. Handler tests. Each handler (photo, PubMed, DOI) will need unit and integration tests.

Possible Future Features

As we gain experience with Simple VIVO and the underlying Pump software, we'll collect ideas for future features here. There are no promises here, and some ideas that seemed worthwhile at the time, may fade in importance as experience grows.

Some of the features are for the "end user," which for Simple VIVO is typically the VIVO data manager. Some are for programmers using the Pump. And some are for the developers of Simple VIVO and the Pump to allow the software to grow and improve in a reasonable way.

Features are numbered to make referring to them easier. The numbering does not imply an ordering.

Note: Some people may ask "why have future features here rather than in the GitHub issues? After all, you can mark issues as "future" and exclude them from lists. And the answer is that I found that mixing the issues together was inconvenient and misleading. GitHub does not have the ability to save queries of the issues, so each time I wanted to see the current issues, I had to explicitly exclude the issues marked "later". Eventually I decided to gather the "later" features here and close them out of the issues list, reserving the issues list for features under active development.

Possible Future Features

1. Add an ontology diagram to each example to show what is going on.
2. Add SPARQL queries and visualizations to each example to reward data managers for getting their data into VIVO.
3. Add handler for photos. When a user specifies a photo filename as data, a handler should process the file, create a thumbnail, put the thumbnail and photo in appropriate places in the VIVO file system and generate the appropriate RDF.
4. Refactor two_step and three_step as recursive. Pump should support any path length through recursive application of path. Need to be clever about "one back" and one forward" and then it should be feasible. Will take a good lucid morning.
5. Simple VIVO should read and write from stdin and stout respectively. Support stdin and stdout as sources and sinks for source data. As in: `filter | filter | python sv.py > file`
6. Simple VIVO should put data directly into VIVO with the need to write a file of RDF to be loaded manually.
7. Add support for auto-generated labels. When adding awards and honors, the VIVO

interface auto generates a label for the AwardReceipt. Without that label, the award receipt displays its uri in the interface. When adding degrees, the VIVO interface does the same thing -- auto generates a label for the AwardedDegree. We could just add a label field to each def and require the data manager to supply the label. And a simple Excel function would auto generate labels in the spreadsheet.

8. Support round tripping of the output of serialize. The Pump serialize method was envisioned as a means of round tripping a text description of the state of the Pump for the purpose of restarting or recreating a Pump. To do this we will need:
    i. A complete description of the Pump state in serialize -- all instance variable values, update_def
    ii. A means to define Pump via the output of serialize, as in `q = Pump.define(p.serialize())` should `define` Pump q to be the same as Pump p.
9. Add an example regarding human subject studies. UF will allow its approved human subject studies to be posted in VIVO and associated with investigators. This is a great resource for understanding scholarship in clinical research. Will need to verify data model (draw a Vue figure, run it through ontology groups), then gather data, map, test.
10. Implement American Universities data management in `uf_examples`. American Universities is a web site at UF used to list "all" accredited universities in the United States offering bachelor's degrees and above. See http://clas.ufl.edu/au
    i. Create separate def for american universities. Web address, name, type
    ii. Add `rdf:type` for american universities to UF VIVO
    iii. Add python script for producing american universities web insert as a UF Example. It uses a UF ontology extension
11. Refactor the CSV object as a true class. Rewrite all CSV access. Class should support column order and empty datasets.

Additional Resources

- VIVO Home Page The VIVO Home Page provide an introduction to VIVO in the form of short videos that highlight the VIVO concepts of connect, share and discover.
- VIVO Project Wiki The VIVO Project wiki is a large repository of community best practices, technical documentation, and project governance.
- VIVO GitHub VIVO software can be downloaded from GitHub. Are you a developer? We are always looking for people to help improve VIVO.
- Some VIVO Things a web site with additional tools, scripts, and SPARQL queries for working with VIVO.

API

An Application Program Interface (API) is a collection of programming routines and definitions for requesting data or actions from a piece of software, and receiving results from software.

DOI

Document Object Identifier. A Unique identifier assigned by the publisher of a work to unqiuely identify the work.

EISSN

Electronic ISSN. The ISSN for the electronic (on-line) version of a journal. See ISSN.

graph

The collection of triples representing the scholarship of an organization can be represented as a graph of nodes and arcs, where nodes represent entities and literal values, and arcs represent relationships between the nodes.

ISNI

The International Standard Name Identifier (ISNI) is a sixteen digit number assigned to persona and organizations in the publishing ecosystem. See isni.org

ISSN

International Standard Serial Number. An eight digit identifier assigned to each journal in the world to uniquely identify that journal. See issn.org

JSON

JavaScript Object Notation (JSON) is a data format that is easy for machines and people to read and write. See JSON.org and the JSON Wikipedia Page

Ontology

"In computer science and information science, an ontology is a formal naming and definition of the types, properties, and interrelationships of the entities that really or fundamentally exist for a particular domain of discourse." *Wikipedia*

VIVO uses several ontologies in common use on the semantic web, as well as VIVO-ISF, an ontology for scholarship, developed for use with VIVO.

Orcid

The Open Researcher and Contributor Identifier. A sixteen digit number assigned to a researcher for the purpose of uniquely identifying that researcher in the scholarly publishing ecosystem. See orcid.org

Python

Python is a popular programming language for scientific, graphical and text programming. See python.org

Pump

The VIVO Pump is python software for adding, removing and updating data in VIVO using spreadsheets.

RDF

Resource Description Framework (RDF) is a way to represent information that is useful for machines. RDF uses triples to represent information.

Semantic Web

The semantic web is an effort to provide open data in standard formats for reuse at web scale.

Triples

RDF represents data using triples of the form

```
subject predicate object
```

Each element of the triple is defined to be a URI or a literal value. Using triples represented using RDF, and ontologies that provide standard meaning for the elements of the triples, VIVO can represent scholarship in a scalable, sharable, well-defined, way.

Triple Store

A triple store is a database designed to manage triples.

URL

Uniform Resource Locator. An address that can be used on the Internet to uniquely identify a resource.

URI

Uniform Resource Identifier. A pattern that can be used to uniquely identify a resource. All URL are URI, but not all URI identify resources on the the network. Some URI are designed for internal or off-line use.

VIVO

VIVO is an open source, semantic web software system for creating and maintaining an integrated view of the scholarly work of an organization. See the VIVO Home Page

VIVO Vagrant

Vagrant is software for providing a self-contained computing environment on your machine. VIVO Vagrant is a full featured VIVO that runs in a Vagrant environment.

VIVO-ISF

The VIVO Integrated Semantic Framework, the ontology used by VIVO to represent scholarship.