Google Refine

Venue:	CAR 2011
	Raleigh, NC
	February 2011
Presenter:	David Huynh, Ph.D.
	Google Inc.
	dfhuynh@google.com
	Twitter @dfhuynh

Outline

1. Key Points
2. Exploration using Facets
   - Facets and Their Interactions
   - Custom Facets
3. Cell Editing
   - Cell Transformations using Expressions
   - In-Facet Editing
   - Clustering
4. Row and Column Editing
   - Batched Row Deletion
   - Duplicate Row Detection and Deletion
5. Structural Editing
   - Transpose Columns into Rows
   - Transpose Fixed Number of Rows into Columns
   - Transpose Variable Number of Rows into Columns
6. Web Scraping (advanced)

Major functionalities NOT covered:
- Connecting to name registries / web databases (a.k.a. reconciliation).
- Calling web services (fetching URLs) to add new columns.
- Loading data into Freebase.

Refer to the screencasts and online documentation of Google Refine for these topics.

For GREL expressions, refer to this tutorial
http://code.google.com/p/google-refine/wiki/UnderstandingExpressions

1. Key Points

Google Refine is a power tool for working with messy data, primarily for

- detecting and fixing inconsistencies
- transforming data from one structure or format to another
- connecting names within your data to name registries (databases)

Use Google Refine when you need something ...

- more powerful than a spreadsheet
- more interactive and visual than scripting
- more provisional / exploratory / experimental / playful than a database

Comparison with Other Tools

Other Tools		**Google Refine**
Spreadsheets	Cells are the units of interaction. Editing happens one cell at a time.	Columns and rows are the primary units of interaction. Editing happens one column at a time, across many rows matching some criteria.
	Used primarily for entering data and performing calculations.	Used primarily for exploring and transforming existing data.
Scripting	You only see the original input data and the final output data.	You see the data visually at each intermediate step as you experiment with editing it and transforming it.
	Used primarily for transforming data.	Used for understanding and then transforming data.
Databases	Serious overhead efforts must be invested in designing schemas.	No schemas required, just like in spreadsheets.
	Data is mostly out of sight unless programming is done to expose views.	Data is always visible, like in spreadsheets.
Adobe Photoshop	Work on pixels.	Work on cells.

How to use Google Refine

- Think in patterns: identify the common characteristics of the cells or rows that you want to change;
- Use facets and filters to isolate them, then invoke a single command to change them all.

Google Refine should work comfortably with 100,000 rows by 10 columns. It is not designed for huge data sets, and it is not designed to replace other tools, but to complement them. Use Google Refine in your larger data processing pipeline in conjunction with other tools.

2. Exploration using Facets

In this section, we will use a relatively clean data set to understand the concept of *facets* in Google Refine. The data set is about Louisiana elected officials, publicly accessible from

    http://www.sos.louisiana.gov/tabid/136/Default.aspx

and available on your workshop station as a file named

    louisiana-elected-officials.csv (1.6Mb, retrieved 01/09/2011)

{1} Launch Google Refine. This should automatically open your web browser to the correct address. If it doesn't, start Chrome or Firefox and browse to this web address: `127.0.0.1:3333`

{2} In Google Refine, on the right-hand side, click Choose File and locate the data file mentioned above. Then click Create Project. (Don't tinker with the import options. The defaults are good.)

{3} Notice that there are 6958 rows total. Rows are shown in pages of 10 each (page size can be changed).

Facets and Their Interactions

{4} Scroll across to see more columns. Locate column named Party Code. Click on its drop-down menu and pick Facet › Text facet. Notice that a *text facet* is opened on the left panel. The text facet reports:

- There are 3700 rows containing D in column Party Code → there are 3700 Democrats.
- There are 15 rows containing N in column Party Code
- ...
- There are 1446 rows having blank cells in column Party Code → lots of party data is missing!

These counts form the *distribution* of the data along the Party Code dimension.

{5} Open another text facet but for column Ethnicity using the same technique.

{6} Looking at the counts in those two facets, you can compute these ratios:

    In Party Code: 1613 R / 3700 D = 0.44
    In Ethnicity: 1255 B / 4469 W = 0.28

Are those ratios in alignment with your knowledge of Louisiana's demographics?

{7} Click on B in facet Ethnicity. Observe:

- The data table on the right is filtered to show only rows containing B in the column Ethnicity (i.e., we're looking at Black officials).
- The count on top of the data table is updated to 1255 matching rows (6958 total), indicating the filtering effect.
- The row numbers on the left of the data table are no longer consecutive (2, 3, 4, 5, 7, 13, 19, 27, ...), also indicating the filtering effect.

If you're familiar with SQL, then the click on B in facet Ethnicity is equivalent to executing this SQL statement

```
SELECT * FROM table_name WHERE Ethnicity = "B"
```

{8} Observe: the counts inside the other facet Party Code have also been updated. These new counts apply to only to the 1255 rows (matching filtering criteria in the other facets). Here is how to interpret the counts:

Among 1255 Black officials:

- 1179 are Democrats
- ...
- 11 are Republicans
- We have no party data for 46 of them.

We can thus infer: almost all Black officials in Louisiana are Democrats.

{9} Click R in the facet Party Code. Now you are seeing the 11 rows that contain B in the column Ethnicity and R in the column Party Code. That is, you are seeing 11 Black Republican officials. The filtering effects of the two facets are compounded, yielding only rows that satisfy both.

An equivalent SQL statement to achieve the same effect is

```
SELECT * FROM table_name WHERE Ethnicity = “B” AND Party_Code = “R”
```

{10} Click reset in both facets (their top right corners) to clear their selections. You should now see 6958 rows again. (You could also have clicked on B in facet Ethnicity and R in facet Party Code to toggle off those selections.)

{11} Open a text facet on the column Office Title. The facet says 76 choices, meaning there are 76 office titles.

{12} Open a *numeric facet* on the column Office Level. Notice the histogram has 4 clusters. Can we conclude that there are 4 strata of officials?

{13} In the numeric facet Office Level, drag the left handle to the right so that the selection window encloses only the 2 right-most vertical bars, thereby filtering for office levels from 300 to 320.

Notice the changes in the other facets. For example, the facet Office Title now has only 21 choices, meaning that for there are supposedly 21 kinds of job titles in the top tier of officials. But that's not entirely accurate, as you might notice that there seem to be 2 choices reading Chief of Police, two reading Council Member, two Councilman and one Councilmen, etc. Hover the mouse pointer over each of the two Chief of Police and notice how their underlines differ:

```
Chief of Police 2
Chief of Police  211
                ↑
```

The second one has a longer underline, indicating a trailing space that has not been trimmed away. This data set is almost clean, but not entirely. We will come back to this issue later.

{14} Click Remove All at the top of the left panel to remove all facets as well as nullifying their filtering criteria. Note that **facets only change which rows you see; they do not modify the data**.

Custom Facets

A facet serves as (a) a summary of the data along one dimension and (b) a mechanism for filtering the data by that dimension. So far, a dimension is just a pre-existing column (e.g., Party Code, Ethnicity, Office Level). But a dimension does not have to be a pre-existing column, but can be computed on-the-fly from some existing columns.

{15} Invoke the drop-down menu on column Phone and pick Facet › Custom text facet ... In the dialog box, in the Expression text field, enter

```
value[0, 3]
```

That is an expression that takes the first 3 characters from the string named `value`. This expression is going to be evaluated on the cell in the column Phone on each row. Before each evaluation, a variable named value is assigned the content of the cell. As each cell in the column Phone contains a phone number starting with the area code, the expression effectively extracts out the area code of each phone number. The dialog box gives a preview of the expression evaluated on the visible rows.

Click OK to create the facet. Click count at the top of the facet to re-sort the area codes by counts. Most people have phone numbers in the 318 area code.

The expression is in a specialized programming language called *Google Refine Expression Language*, or *GREL* for short. It has some similarities with Excel formula language and Javascript.

{16} Numeric facets can also be customized. For example, the column Commissioned Date currently contains strings, which can be parsed into dates. To create a numeric facet for commissioned **years**, invoke the drop-down menu on the column Commissioned Date and pick Facet › Custom numeric facet ..., then enter the expression

```
value.toDate().datePart("year")
```

Notice that you could have written that same expression as

```
datePart(toDate(value), "year")
```

which is more similar to how Excel formulas look. I personally prefer the first form, as it more clearly communicates the order of function calls (i.e., `toDate` first, then `datePart`).

{17} As another example of custom numeric facets, consider the column Office Description, which often contains something like “1st Representative District ...”. We can create a numeric facet on that column to filter by which district an official is from, using this expression:

```
value.match(/\D*(\d+)\w\w Rep.*/)[0].toNumber()
```

That contains a *regular expression* and is not for the faint of heart, but it gives you a peek into what you can do with GREL expressions.

(We will continue using this project in the next section.)

3. Cell Editing

(We will continue using the same project as the last section.)

Cell Transformation using Expressions

GREL expressions can be used to create custom facets as well as transforming data. For example, look at column Zip Code 2: some zip codes have 5 digits and are parsed by Google Refine into numbers, while others with 9 digits are left as strings. Let us convert them all to 5-digit strings.

{1} We will continue with the same project from the last section. Remove all facets from the left panel if you have any.

{2} Invoke the drop-down menu of column Zip Code 2 and pick Edit cells › Transform ... Then enter this expression

```
value.toString()[0, 5]
```

and click OK. Notice a message at the top of the window about 6067 cells having been changed.

{3} On the left panel near the top, click on the tab Undo/Redo. This tab shows you all editing operations that you have done (and possibly undone) to the data, all the way back to the beginning when you just created the project. This *history* is saved with the project and does not get lost even if you close your browser and/or shut down Google Refine.

The dark blue operation in the history is the last performed operation--the one that yields the current state of the data. You can click on any operation in the history to make it the last performed operation, thus effectively undo or redo other operations.

{4} Remember the issue we discovered previously about office titles with trailing spaces? Let's fix that problem. Open a text facet on the column Office Title and notice that it has 76 choices. Then on that same column, invoke its drop-down menu and pick Edit cells › Common transforms › Trim leading and trailing whitespace. Now notice that the facet only has 67 choices.

In-Facet Editing

The problem of Councilman vs. Councilmen has not been solved by space trimming. Suppose we decide that Councilman is the right form, then we need to change all occurrences of Councilmen into Councilman.

{5} Scroll inside the facet Office Title until you locate Councilman and Councilmen. Note that there are 281 of the former, and 13 of the latter. Hover your mouse over Councilmen until you see edit to its right. Click on edit, and type in Councilman, and hit Enter. Now you see that the count next to Councilman has been changed to 294 (281 + 13). You have just done a whole-cell search and replace operation that changed 13 cells.

Clustering

{6} Councilman vs. Councilmen is probably just one case of a more general problem of spelling variations. To detect the other cases, click the button Cluster in the facet Office Title.

{7} This clustering feature attempts to group facet choices together based on some similarity metrics using various methods. The first method key collision does not detect any cluster, so click on the drop-down Method and pick nearest neighbor. Now you should see some clusters.

Some clusters show clearly different choices (e.g., RSCC Member vs. DSCC Member) while others show choices that should be the same (e.g., Council Member vs. Councilmember). Check the checkboxes next to the two clusters whose choices should be the same, and then click the button Merge Selected & Close at the bottom of the dialog box. That solves the whole family of spelling variation problems.

4. Row and Column Editing

Batched Row Deletion

Let's continue from the last task of the previous section, which used clustering to fix data inconsistencies. Sometimes clusters need more special treatment than just simple "merging" (replacing all variations with a single form). In this section, we look at inconsistencies that look like duplicated data, which call for row deletion.

{1} On the column Candidate Name, invoke the menu Edit cells › Cluster and edit (the feature doesn't have to be invoked through a facet only). This time, the method key collision can find some clusters, many of which have one choice with a trailing comma, e.g.,

```
Wayne Landry,
            ↑
```

That looks like a data processing error.

{2} Hover the mouse over the cluster containing Wayne Landry until you see a link Browse this cluster, then click it. This opens up a new tab showing just 1 rows, one containing Wayne Landry and one containing Wayne Landry, (note the trailing comma). Their row indices are consecutive, so perhaps the second entry was a mistake and needs to be removed. Click on the star icon in front of that row to mark it.

{3} In the first web browser tab that still contains the Cluster & Edit dialog box, go through some of the other clusters with the trailing comma problem and star the rows to be removed.

{4} Then return to the first tab and click Close to dismiss the Cluster & Edit dialog.

{5} Click on the drop-down menu next to All (top left corner of the data table) and pick Facet › Facet by star. Note that All means the menu applies to the data set in general, not to any column in particular. And the facet is just a predefined customized facet with the expression:

```
row.starred
```

{6} Select true in the new facet. Then in the same drop-down menu of All, pick Edit rows › Remove all matching rows. Then remove the facet. You have removed all those extraneous rows that you have starred previously in one shot.

The command Remove all matching rows also works with any other facet, filter, or a combination of several facets and or filters. As long as all the rows you want to remove exhibit a common pattern, and other rows do not, then you can create facets or filters that constrain for rows matching that pattern and then invoke Remove all matching rows.

Duplicate Row Detection and Deletion

Sometimes we detect redundant or duplicate rows not to delete them, but to aggregate them.

{7} Click on the Google Refine logo at the upper left corner to return to the Google Refine home page and create a new project using the file named `duplicates.csv`.

This data set contains data about customers. We want to count the number of times each customer made purchases, and each customer is identified uniquely by his or her email address. A customer might have made several purchases, thus appearing on multiple rows, but the rows don't have to be consecutive. We must first sort the rows so that those rows about a single customer would be next to one another. Then for each such group of consecutive rows about a single customer, we will eliminate all except the first row in the group.

{8} Invoke the drop-down menu of the column email and pick Sort .... Just click OK in the Sort dialog box. Now note how the indices of the rows are not in consecutive order anymore.

{9} Sorting changes the order in which the rows are shown to you, but the order in which they are stored has not been changed. To change their stored order, look for Sort ▾ just above the data table, somewhere in the middle, and pick Reorder rows permanently. Now note that the rows are indexed consecutively.

{10} Now we want to count how many times each email address occurs. Invoke the drop-down menu of the column email and pick Edit column › Add column based on this column.... Name the column count and enter this expression

```
facetCount(value, "value", "email")
```

and click OK. You should see 2 next to arthur.duff@example4.com, 3 next to danny.baron@example1.com, and so forth.

{11} In the drop-down menu for column email, invoke Edit cells › Blank down. This command blanks out subsequent consecutive duplicated cells. For example, previously there were 3 instances of danny.baron@example1.com on rows 5, 6, and 7, and after invoking the command, only the first instance on row 5 remains. The other two cells have been blanked out.

{12} In the drop-down menu for column email, invoke Facets › Customized facets › Facet by blank. This is just a predefined customized facet with the expression:

```
isBlank(value)
```

{13} Select true in the facet, invoke the command Remove all matching rows in the drop-down menu of All, and then remove the facet. You should now have 6 rows with unique email addresses as well as the count of each occurring in the original data.

5. Structural Editing

The previous section shows some very basic capabilities for row and column editing. Often, you need to perform more invasive surgery on the data than just removing rows or adding columns. In this section, we will transpose columns into rows and rows into columns, and also add and remove rows and columns as necessary.

Transpose Columns into Rows

{1} Create a new project using the file named `us_economic_assistance.csv`. This data was originally downloaded from

http://www.data.gov/raw/1554

and contains data about economic assistance given by the U.S. to other countries. It is also known as the "green book".

Each row contains data for a combination of country and program, spreading over many years from 1946 to 2008. Consequently, there are many columns.

Let us assume that we want to load this data into a SQL database, and we have a table of 4 columns: country, program, year, and amount. We need to execute SQL statements in the form of

```
INSERT INTO table_name (country, program, year, amount) VALUES ( ... )
```

and the data as it is currently structured does not lend itself to formulating these SQL statements. We must transform it, from this structure

country_name	program_name	FY2006	FY2007	FY2008
Afghanistan	Child Survival and Health	37974000	69444000	27813000

into into this structure

country_name	program_name	amount	year
Afghanistan	Child Survival and Health	37974000	2006
Afghanistan	Child Survival and Health	69444000	2007
Afghanistan	Child Survival and Health	27813000	2008

Note that the dashed cells have been effectively rotated 90 degrees clockwise, or **transposed** from a single row across 3 **columns** into 3 **rows** in a single column. Also note that the years get from the column headers into the rows as data.

{2} Invoke the drop-down menu of the column FY1946 (the left-most column that needs to be transposed) and pick Transpose › Cells across columns into rows.... In the dialog box, note that the first list has FY1946 selected as the first column to transpose, and the second list has FY2008 selected as the last column to transpose.

{3} Enter `pair` for the combined column name. Make sure that prepend column name is checked, and the separator is a colon. Then click Transpose.

Now we have a column called pair and each of its cells contains a pair of year and amount. We need to extract out the years and the amounts into different columns.

{4} Invoke the menu of the column pair and pick Edit column › Add column based on this column.... In the dialog box, set the new column name to `year` and enter the expression

```
value[2,6].toNumber()
```

Then click OK.

{5} Invoke the menu of column pair again and pick Edit cells › Transform.... In the dialog box, enter the expression

```
value.substring(7).toNumber()
```

Then click OK.

{6} Invoke the menu of column pair again and pick Edit column › Rename this column and enter `amount`.

{7} For each of the two columns country_name and program_name, invoke their drop-down menu and pick Edit cells › Fill down. Now the data is in a structure that can easily be transformed into those SQL statements mentioned earlier.

Transpose Fixed Number of Rows into Columns

{1} Return to the Google Refine home page. Make sure that the checkbox Split into columns is NOT checked, and enter `0` into the text field Header lines. Then set the Data file field to the file named `fixed-rows.csv` and create a new project.

Take a look through the data: it is about money transactions. Some lines are about money being sent, and some are about money being received. When money is sent, the address provided is of the recipient; when money is received, the address is of the sender.

We would like to transform

Column
Tom Dalton sent $3700 to Betty Whitehead on 01/17/2009
377 El Camino Real
San Jose, CA
Status: received

into

Transaction	Sender	Recipient	Amount	Date	Address	Address 2	Status
send	Tom Dalton	Betty Whitehead	$3700	01/17/2009	377 El Camino Real	San Jose, CA	Status: received

This involves transposing cells in those 4 rows into columns as well as parsing out the first cell into several columns.

{8} Invoke the menu of column Column and pick Transpose › Cells in rows into columns.... Enter `4` and click OK. You should now have 4 rows and 4 columns.

~~{9} Optionally, rename Column 2 to Address, Column 3 to Address 2, and Column 4 to Status.~~ (Skip this step: it triggers a bug in Google Refine.)

{10} Let us now create the column called Transaction. Invoke the drop-down menu of column Column 1 and pick Edit column › Add column based on this column.... Set the new column name to `Transaction` and enter this expression

```
if(value.contains(" sent "), "send", "receive")
```

and click OK. The expression should be quite self-explanatory.

Note that the order of the sender and recipient on each row in Column 1 depends on the transaction:

> Sender A sent $ to Recipient B on ...
> Recipient C received $ from Sender D on ...

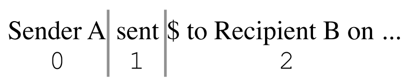
We will need to process the two cases separately. First, we'll address the sending case.

{11} Create a text facet on the column Transaction and select send in it.

{12} Add a column based on Column 1 using this expression

```
value.partition(" sent ")[0]
```

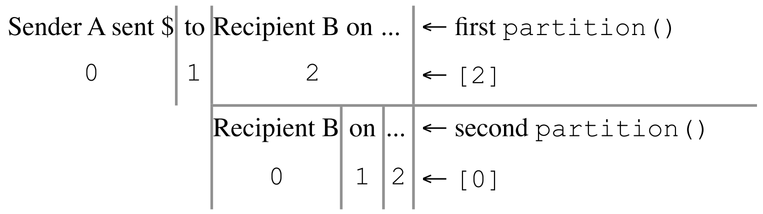
and name it `Sender`. The function `partition()` splits the string value into an array of 3 string segments, with the middle segment equal the string `" sent "` (the leading and trailing spaces are important). `[0]` returns the first segment in that array.



{13} Add another column based on Column 1 using this expression

```
value.partition(" to ")[2].partition(" on ")[0]
```

and name it `Recipient`. Here is how to picture the expression:



{14} Add yet another column based on Column 1 using this expression

```
value.partition(" sent ")[2].partition(" to ")[0]
```

and name it `Amount`.

{15} Click receive in the facet.

Note that the cells in columns Sender, Recipient, and Amount are blank. Rather than creating more columns, we're just going to perform cell transformations on these three existing columns to fill them in.

{16} Perform cell transformations on the columns Sender, Recipient, and Amount using the following expressions, respectively:

```
cells["Column 1"].value.partition(" from ")[2].partition(" on ")[0]
cells["Column 1"].value.partition(" received ")[0]
cells["Column 1"].value.partition(" received ")[2].partition(" from ")[0]
```

Note that `cells["Column name"].value` is how we can refer to the content of the cell in another column on the same row.

{17} Remove the facet so that you now see all 4 rows again, all filled in.

{18} Perform the following cell transformation on column Column 1

```
value.partition(" on ")[2]
```

and rename the column to `Date`.

{19} Optionally, invoke the drop-down menu in front of All and pick Edit column › Reorder columns.... In the dialog box, drag and drop the columns to whatever order you desire.

Transpose Variable Number of Rows into Columns

It is easy to transpose cells in rows into columns if every record occupies the same number of rows (4 in the previous scenario). It is trickier if each record occupies a different number of rows. To transpose cells in a variable number of rows into columns, we will use the concept of *record* in Google Refine.

{20} Create a new project using the file `variable-rows.csv` and be sure to uncheck the checkbox Split into columns and set the field Header lines to 0.

This data set looks similar to the previous one, but for some records there are phone numbers as well. So the number of rows per record can be 4 or 5. This is what makes it tricky.

{21} First, we are going to create a column that identifies the first row of each record. Invoke the drop-down menu of the only column Column and pick Edit column › Add column based on this column.... Name it `First Line` and enter this expression

```
if(value.contains(" on "), value, null)
```

Click OK. Note how only the first row of each record is copied over to the new column.

{22} Invoke the drop-down menu of column First Line and pick Edit column › Move column to beginning.

{23} Just above the table on the left, find

Show as: **rows** records

and click on records. Observe that the first 4 rows now share the same background color and indexed together as 1. The next 5 rows also share the same background color (but different from the first 4 rows) and are indexed together as 2. And so forth. In other words, rows of each record have been grouped together. Feel free to toggle between rows and records to see the difference, but at the end, leave it in records mode.

Google Refine groups rows into records simply by looking at the left-most column. Rows blank in the first column get grouped with the first preceding row that is not blank in the first column.

Now an expression evaluated on a row can refer to other rows in the same records.

{24} Create a new column based on the column named Column, name it Status and enter this expression

```
row.record.cells["Column"].value[-1]
```

Note that

```
row.record.cells["Column"]
```

refers to all cells in column Column in all rows of the current record. The result is an array of cells. Going one step further,

```
row.record.cells["Column"].value
```

refers to the content of those cells, as an array of values (strings in this case). `[-1]` picks out the last value, which always corresponds to the status.

{25} Next, perform a cell transformation on the column Column using this expression

```
row.record.cells["Column"].value[1, -1].join("|")
```

That's a pipe character, not a number one or a lowercase L or an uppercase I. The indices 1 to -1 select elements in the array starting from the second element up to but excluding the last element. The function `join()` concatenates the strings in the resulting sub-array with the given separator in between them.

{26} Switch back to rows mode.

{27} Create a Facet by blank on column First Line, select true in it, invoke Remove all matching rows, and remove the facet.

{28} Invoke the menu of column Column and pick Edit column › Split into several columns.... Specify the pipe character `|` as the separator and click OK.

It should be obvious how to process this data further.

6. Web Scraping (advanced)

It is not too rare that you would find data on the web locked in PDF files. Messy CSV, TSV, Excel files are already hard to process, but PDF files add yet another level of difficulties. Fortunately, ScraperWiki http://scraperwiki.com/ has a PDF to XML converter, and when it is used in conjunction with Google Refine, we have a free tool chain for scraping data tables out of multi-page PDF files.

Let us scrape one of the data sets that ProPublica used for their "Dollars for Docs" report
    http://projects.propublica.org/docdollars/
ProPublica gives us screenshot previews of those data sets here
    http://projects.propublica.org/docdollars/payment_reports
We're going to pick Eli Lilly as an example, which leads us to
    http://www.lillyfacultyregistry.com/pages/lilly-registry-report.aspx
It is a Flash application, but there is a link labeled Download this Report that yields this PDF file
    http://www.lillyfacultyregistry.com/documents/EliLillyFacultyRegistryQ22010.pdf

Take a moment to look over that file yourself, and note:

- There are 71 pages in total. So it's not easy to copy and paste all the data into another application. The PDF reader does not allow text selection that crosses from one page to another.
- The tables on the pages look very consistent in that their columns all line up across pages. This consistency is really important.
- Except for the very first page, all pages have the same header, meaning that their data rows start at the same offset from the tops of the pages, and end at the same offset from the bottoms. This consistency is also really important.
- Numbers are right-aligned in each column, meaning that they start at different offsets from the left edge of the page, even if they are in the same column. Fortunately, they are always bounded by the left edge of the column, so there is still a consistency.

There is already a Python-based "scraper" on ScraperWiki that can process that PDF file:
    http://scraperwiki.com/scrapers/eli-lilly-dollars-for-docs-scraper/edit/
Its source code is shown here

```
import scraperwiki
import urllib
import lxml.etree, lxml.html
import re

pdfurl = "http://www.lillyfacultyregistry.com/documents/
EliLillyFacultyRegistryQ22010.pdf"

pdfdata = urllib.urlopen(pdfurl).read()
pdfxml = scraperwiki.pdftoxml(pdfdata)
root = lxml.etree.fromstring(pdfxml)

for page in root:
    assert page.tag == 'page'
    pagelines = { }
    for v in page:
        if v.tag == 'text':
            text = re.match('(?s)<text.*?>(.*?)</text>',
                      lxml.etree.tostring(v)).group(1)
            top = int(v.attrib.get('top'))
            if (top - 1) in pagelines:
                top = top - 1
            elif (top + 1) in pagelines:
                top = top + 1
            elif top not in pagelines:
                pagelines[top] = [ ]
            pagelines[top].append((int(v.attrib.get('left')), text))
    lpagelines = pagelines.items()
    lpagelines.sort()
    for top, line in lpagelines:
        line.sort()
```

```
            key = page.attrib.get('number') + ':' + str(top)
            scraperwiki.datastore.save(unique_keys=[ 'key' ],
              data={ 'key' : key, 'line' : line })
```

You don't have to understand this Python code. Just know that it is quite generic, meaning that if you replace the URL pointing to that PDF file with a URL pointing to another multi-page PDF file containing tables of data, the scraper would also produce useful result.

The text content of the PDF file is stored as a collection of little text fragments. For example, the piece of text in each cell is a text fragment. Each text fragment has a pair of coordinates relative to the top left corner of the page. Text fragments in the same table row have (almost) the same "top" offset; and text fragments in the same table column have (almost) the same "left" offset (unless they are right-aligned). I say "almost" because these offsets can sometimes be off by 1, which is why you see the +1 and -1 tests in the Python code.

The output of the scraper is a table with two columns, key and line. Here is a sample row:

```
71:95       [(467, '<b>Programs</b>'), (532, '<b>Education&#160;Programs</b>')]
```

Each row is about text fragments on one line on one page. The key is made up of the page number followed by the "top" offset of the line. This sample row is about a line on page 71, starting at 95 units from the top of the page. We don't care what the unit is--cm or inch or pica, we just need those numbers to sort the rows. We will also use the "top" offsets to eliminate everything except the data rows.

The second column contains the text fragments as well as their "left" offsets. So, this line has 2 text fragments: "Programs" starting at offset 467 and "Education Programs" at 532. `&#160;` is a space encoded. We will use these "left" offsets to pick out the cells to form separate columns.

On the scraper page
    http://scraperwiki.com/scrapers/eli-lilly-dollars-for-docs-scraper/
there is a link labeled Download this dataset (CSV) to would yield the data, but this tutorial already includes one version of the data set for you.

{1} Create a new project from the file `eli-lilly.csv` (use all default importing options).

{2} Split the column key by a colon. Rename column key 1 to page and key 2 to top. And move column line to the right most position.

{3} Sort column page as numbers, smallest first, and then also sort column top as numbers, smallest first. Reorder the sorted rows permanently.

Looking at page 2 of the PDF file, we find Ahman, Andrew J to be the first name and Alvarez, Ronald D to be the last name.

{4} Create a text filter on column line and type `ahman` into it. We get exactly one row, whose top cell contains 106. Change the text filter to `alvarez` and note that the last line's "top" offset is 567. Remove the text filter.

{5} Create a numeric facet on column line. Select range 0 to 100 in the facet and remove all matching rows (775 of them). Select range 570 to 600 and remove all matching rows (71 of them). Reset the facet but don't remove it.

Since the first page has a taller header, we need to treat it separately.

{6} Create a text facet on column page and select 1 in it.

{7} Note that the first name on page 1 is 4 Wellness R.D., Inc, which starts at offset 161. So select range 20 to 160 in the first facet, and remove all matching rows. Now remove all the facets. Now only the data rows remain.

{8} Next, clean up the column line by doing a cell transform using this expression

```
forEach(
  value[2,-2].replace("&#160;", " ").split("), ("),
  v,
  v[0,-1].partition(", '", true).join(":")
).join("|")
```

This fixes the &#160; issue, removes single quotation marks around the text fragments, separates the “left” offsets from the text fragments with colons, and separates the text fragments with pipe characters.

The `forEach` construct is new here. It is to evaluate the same sub-expression on each element of an array.

{9} Now we notice that cells in the column “Name” always start at “left” offset 16. So create a column based on column line using this expression:

```
filter(
  value.split("|"),
  p,
  p.partition(":")[0].toNumber() == 16
)[0].partition(":")[2]
```

and name it `Name`.

The `filter` construct is new here. It is used to filter an array for only elements satisfying a given condition.

{10} Similarly create a column called Location using the same expression but with offset 236 rather than 16. You can use the tab History in the dialog box to recall the previous expression.

{11} Also create a column called State with offset 322.

{12} Also create a column called Provider of Service with offset 347.

{13} Now that we have extracted out all 4 left-aligned fields, we can simplify the column line by doing the following cell transform:

```
filter(
  value.split("|"),
  p,
  p.partition(":")[0].toNumber() > 347
).join("|")
```

What’s left are the 5 right-aligned columns.

{14} Create a custom numeric facet on the column line with this expression

```
forEach(value.split("|"), v, v.partition(":")[0].toNumber())
```

This expression picks out all the “left” offsets. The histogram clearly shows 5 lumps, corresponding to the 5 columns. Move the range selectors of the facet around and watch the range numbers updated below the histogram. We can say that the 5 columns correspond to these offset ranges:

- “Patient Education Programs”: < 500
- “Healthcare Professional Education Programs”: 500 - 540
- “Advising/Consulting & International Education Programs”: 550 - 590
- “Certain Travel-Related Expenses”: 590 - 640
- “2010 To Date Aggregate”: > 670

{15} Remove the numeric facet.

{16} Create a column called Patient Education Programs based on the column line using this expression:

```
filter(
  forEach(
    value.split("|"),
    v,
    v.partition(":")
  ),
  a,
  with(a[0].toNumber(), left, left < 500)
)[0][2]
```

{17} Similarly, create a column called Healthcare Professional Education Programs using this expression:

```
filter(
  forEach(
    value.split("|"),
    v,
    v.partition(":")
  ),
  a,
  with(a[0].toNumber(), left, and(left > 500, left < 540))
)[0][2]
```

Note that only the comparison has been changed.

{18} Do the same for the remaining 3 right-aligned columns.

{19} Transform cells in each of those 5 columns into numbers

```
value.replace(":", "").toNumber()
```

This web scraping exercise is intended to illustrate what "thinking in patterns" means. The patterns might not reside within the data itself, but might be in how the data is presented. Offsets, fonts, colors, etc. can all be used to parse out the data if such presentation information can be recovered and fed into Google Refine.