Datasets

Social scientists often work with datasets that are structured as a collection of records, where each record consists of multiple fields, and each field has an identifier and a value. Such data is in a *tabular format* (or more simply, it is a table). Abstractly, this structure is naturally modeled as a list of associations, where each association represents a single record, and where the keys of the association are the field identifiers. This section briefly considers a WL data type that can be very similar to a list of associations: the **Dataset**. It provides the same natural representation of tabular datasets, with additional advantages for transformation, selection, and aggregation. (For more details see the Wolfram Language Guide entitled Computation with Structured Datasets.)

Creating Datasets

The **Dataset** command can turn a list of associations into a unified structure that is easily queried and manipulated. Such a dataset is essentially just a convenient wrapper around a list of associations. (When needed, use the **Normal** command to reverse this process, creating a list of associations from such a dataset.) As an additional feature, in a Mathematica notebook such a dataset should have a convenient visual display.

Out[160]=

Expression	Result

```
ds01=Dataset[{
    assoc01,
    assoc02,
    assoc03
}]
```

first	last	h1	h2
John	Doe	50	30
Jane	Doe	30	50
Signor	Rossi	46	45

Most often, dataset creation involves import from an external file. The **SemanticImport** command excels at creating data sets from external data files, such as CSV files. (The WL documentation provides extensive details.) In order to produce a self-contained example, the following illustration uses the closely related **SemanticImportString** command, which operates directly on strings.

Out[•]=

Expression	Result

```
SemanticImportString[
"first,last,h1,h2
John,Doe,50,30
Jane,Doe,30,50,
Signor,Rossi,46,45"]
```

first	last	h1	h2
John	Doe	50	30
Jane	Doe	30	50
Signor	Rossi	46	45

Filtering and Recoding Datasets

Criterion-based filtering of datasets is a very common need. From a given dataset, researchers often work with a smaller dataset comprising only those records that satisfy some criterion. Recall that this process is called *selective set building*, *filtering*, or *subsetting*. The **Select** command can perform dataset filtering: apply it to a dataset and an appropriate predicate (i.e., a function that returns **True** iff a record satisfies our criterion). For example, from **data01** keep only those records where the **h1** and **h2** fields sum to less than 90. Recalling that when associations that have strings as keys, functions operating on them can refer by key name to the values, proceed as in the following example. This illustrates the operator form of **Select**, which can be particularly convenient when the same criterion will be used to subset several datasets. (See the documentation for details.) Note that this would work just well with a list of associations, although the display would not be as nice.

Out[•]=

Expression	Result

```
select01=Select[#h1+#h2<90&];
select01[ds01]
```

first	last	h1	h2
John	Doe	50	30
Jane	Doe	30	50

Substitutive set building is sometimes called *transformation* or *recoding*. For datasets, the **Map** command (discussed in Chapter 2) largely covers substitutive set building. One may map over the records of a dataset to produce a new dataset. Typically, dataset transformations retain some kind of unique record identifier, precluding the creation of exact duplicates. In a WL dataset, the records are ordered, and that order is retained after mapping.

Recall from Chapter 2 that functions transform associations are allowed direct access to the association keys. For example, the following expression defines a function **sub01** that accesses four fields of an association and then creates a new association.

This facility implies that **Map** can create recoded datasets with easily understood transformations. For example, recalling the slash-atmark (/@) shorthand for **Map**, the following example produces a new

dataset that holds only names and totals for each record of the original dataset. To understand this, think of a dataset as essentially a list of associations, so that this code essentially maps across that list and produces a new list of associations. In fact, one may proceed in exactly this way, sacrificing only the convenient display of the result.

```
Expression      Result
sub01 /@ ds01
```

Out[•]=

first	last	total
John	Doe	80
Jane	Doe	80
Signor	Rossi	91

Queries for Recoding and Selection

In database terminology, selective or substitutive set building performs a *query*, which retrieves information from a dataset. WL correspondingly provides the **Query** command for producing query operators. Applying a query operator to a dataset produces a subset or a transformation (or a transformed subset). The query syntax is extremely flexible, and queries may be complex. This subsection illustrates a few simple queries.

As initial example of queries, consider the filtering and recoding examples from the previous subsection. The query to implement the filtering example is **Query[select01]**. The query to implement the recoding example is **Query[All,sub01]**. Each of this produce a function that may be applied to a data set (or a list of associations) to produce the associated filtering or recoding. In addition, they can be readily combined, as follows. After selecting only the records of interest (rather than all of them), each record is transformed.

```
Expression                      Result
Query[select01,sub01] @ ds01
```

Out[•]=

first	last	total
John	Doe	80
Jane	Doe	80

There is another useful way to implement the same query. As a convenient shorthand for such a query, a data set can be applied directly to the query components.

```
Expression              Result
ds01[select01,sub01]
```

Out[•]=

first	last	total
John	Doe	80
Jane	Doe	80

As another example of recoding a dataset, consider the selection of a subset of the fields from every record. First characterize set of records to process (here, all of them), and then list the desired columns (by name). Here instead of applying **Query[All,{"h1","h2"}]** to the dataset, use the specialized query syntax from the previous example. (An alternative query to achieve this result is **KeyDrop[{"˙. first","last"}]**.)

```
Expression              Result
ds01[All,{"h1","h2"}]
```

Out[•]=

h1	h2
50	30
30	50
46	45

Recall again that functions applied to associations can make direct use of the association keys, which facilitates easy creation of transformed datasets with **Map**. For example, the following function will append to an associations a new key (**"total"**) with a value that is the sum of the values associated to the **"h1"** and **"h2"** keys.

It is quite possible to map this function across the data01 dataset. In this fashion, substitutive set building can add fields. When recoding datasets, a more idiomatic alternative is to create an equivalent query: once again use **Query** with the transformation as the second argument. The following example again does this by means of the specialized query syntax; it use **sub02** to append a **"total"** column to the dataset.

```
Expression                Result
ds02 = ds01[All,sub02]
```

Out[•]=

first	last	h1	h2	total
John	Doe	50	30	80
Jane	Doe	30	50	80
Signor	Rossi	46	45	91

As noted before, the **Query** command is remarkably flexible. For example, as explored in the next subsection, queries can sort the data set on arbitrary criteria. An interesting and surprising application of dataset queries is that **LinearModelFit** can be used as a query operator, after removing the field names. To discard the headers, use the **Values** command. This produces a headerless rectangular dataset, which is essentially a rectangular list of lists rather than a list of associations. (Use the **Normal** command to produce the underlying list of lists.)

```
Expression                                    Result
hvals=Values@ds01[All,{"h1","h2"}];           FittedModel[ 75.4167 – 0.803571 x ]
    hvals[LinearModelFit[#, x, x]&]
```

Out[•]=

Later chapters discuss linear regression in more detail. The Wolfram Language *How To* entitled Perform a Linear Regression provides a useful introduction.

Partitioning and Classification

A *partition* of a set *X* is a collection of non-empty, pairwise-disjoint subsets whose union is *X*. Each of these subsets is called a *block*. Recall that a collection of sets is *pairwise-disjoint* if the intersection of any pair of the sets is empty, so the blocks of a partition have no elements in common. The simplest interesting partition is a two-block partition. Two-block partitions are naturally formed by dividing set elements into those that satisfy a certain property and those that do not. Whenever selective set building picks a subset of items that satisfy some predicate, one may produce a corresponding partition by additionally creating the complement of that subset.

Since a list without duplicates may nicely represent a finite set, partitioning a list can illustrate partitioning a set. The following example uses **TakeDrop** and **Partition** to produce some arbitrary blocks from a small list of numbers. The basic form of the **TakeDrop** command takes as arguments a list and an initial sequence length. It partitions the list into two sublists: an initial sequence, and the remainder of the list. The second argument can alternatively be a list of two integers, which specify the start index and stop index of a sublist to take. The **Partition** command offers a different approach to partitioning: it can partition a list into sequential sublists of a common length. The following example uses the two-argument form of **Partition**, where the first argument is a list and the second argument is the length of the sublists. In order to avoid discarding terminal elements when the list is not evenly divisible, use the

UpTo command in the second argument.

Out[•]=

Expression	Result
n7=Range[7]	{1, 2, 3, 4, 5, 6, 7}
TakeDrop[n7,3]	{{1, 2, 3}, {4, 5, 6, 7}}
TakeDrop[n7,{3,5}]	{{3, 4, 5}, {1, 2, 6, 7}}
Partition[n7,UpTo[3]]	{{1, 2, 3}, {4, 5, 6}, {7}}

This example shows that, given a list that represents a finite set, **TakeDrop** or **Partition** can produce a list of lists that represents a particular set partition. However, social scientists more commonly need a partition that is based on properties of the elements. Such a partition is called a categorization or a *classification* of the elements, where the classification typically derives from a classifier function.

Use of GroupBy

The simplest two-argument form of the **GroupBy** command accepts as arguments a list to be classified and a classifier function. The result is an association from categories in the range of the classifier function to the lists of elements in each category, representing a *one-way classification* of the data. For example, a set of households might be classified by household size. As a simple illustrative example, consider the partition of a set of integers into two subsets that are the even members and the odd members. The following functions serves as classifier examples. The first maps each integer to a single classification, either **"even"** or **"odd"**. The second maps each integer to a single classification, either **"div3"** or **"¬div3"**; this classifies any integer as divisible by 3 or not.

When applied to a set of elements, categorization with **GroupBy** represents the inverse relation of the classifier function on a set. For example, apply the **div2** classifier to the first seven positive integers, as follows.

Out[•]=

Expression	Result
gb2=GroupBy[n7,div2]	⟨|odd → {1, 3, 5, 7}, even → {2, 4, 6}|⟩

In a Mathematica notebook, applying the **Dataset** command to the resulting association produces a convenient tabular display.

Out[•]=

Expression	Result
ds2 = Dataset[gb2]	

odd	{1, 3, 5, 7}
even	{2, 4, 6}

As a second simple example, use the one argument form of **GroupBy**, which produces a grouping operator. (Recall that the at-sign operator is the prefix notation for function application.)

```
Expression                          Result
Dataset @ GroupBy[div3] @ n7
```

Out[•]=

¬div3	{1, 2, 4, 5, 7}
div3	{3, 6}

A single classification can partition a dataset. A second classification can then refine this partition. That is, partition blocks can themselves be partitioned, resulting in a refined partition of the original set. (Partition Π' is a refinement of partition Π iff every block of Π' is included in a block of Π.)

It is possible to sequentially refine a partition by mapping a second **GroupBy** operator across the values of an initial grouping. However, the **GroupBy** command simplifies this: it accepts a list of classifiers that are sequentially applied. The following example illustrates this nested categorization by producing a *two-way classification*. The result is a multilevel association: an association whose values are in turn associations. Once again, the **Dataset** command can produce a convenient tabular display of the result. The outer association keys are used as row headers. When the inner associations share keys, as they do here, the **Dataset** command typically displays these keys as column headers. (Refined control of the display of datasets is beyond the scope of this book, but see below for a few hints.)

```
Expression                          Result
ds23 = Dataset @
    GroupBy[{div2,div3}] @ n7
```

Out[•]=

	¬div3	div3
odd	{1, 5, 7}	{3}
even	{2, 4}	{6}

Block Size as a Dataset Query

Social scientists often care about the sizes of the partition blocks. The sizes of blocks produced by categorization is a frequency count for the categories. A one-way frequency table displays categorical data as a list of categories and their frequency of occurrence in a dataset. Mapping **Length** across a one-way categorization produced by **GroupBy** will produce the information for a one-way frequency table (as an association from categories to counts). However, when working with data set objects, it is most natural to produce the length of each block by means of a query. Therefore, this section instead uses a dataset query to accomplish the same goal. Construct the query with the following reasoning: for each category in a one-way categorization, we want the total number of items in that category. This suggests the query **Query**[**All**,**Length**]. Try this out on the previous even-odd classification dataset. In a Mathematica notebook, the result displays as a simple one-way table.

Out[•]=

Expression	Result
ds2[All,Length]	

odd	4
even	3

The same approach can create a two-way frequency table, which shows frequency counts for each possible pair of classifications. Recall that a two-way categorization using **GroupBy** creates a multi-level association, which is an association whose values are also associations. A frequency table reports the number of items for each value of the inner association. This suggests the query **Query[All,All,Length]**. Try this out on the previous two-way classification dataset. In a Mathematica notebook, the result displays the element count for each block of the resulting partition of the dataset. It is a frequency table for data described by two categorical variables, which is often called a two-way table, a two-dimensional cross-tabulation, or a contingency table.

Out[•]=

Expression	Result
ds23[All,All,Length]	

	¬div3	div3
odd	3	1
even	2	1

The **Map** command can also produce this result, by means of the three-argument form. (The optional third argument specifies the level at which to do the mapping.) This again displays the power of **Map**; nevertheless, the use of queries is idiomatic when working with datasets.

Out[•]=

Expression	Result
Map[Length,ds23,{2}]	

	¬div3	div3
odd	3	1
even	2	1

Display Issues

The example above produces a nice cross-tabulation display with little effort. In the presence of empty categories, a nice display involves a bit more work. For example, pick a different secondary classifier: divisibility by 4 instead of divisibility by 3. The result is still a partition of the set, but now nothing is listed for the empty category (odd and divisible by 4). Using the **Dataset** command to produce a convenient categorical display is still acceptable, but the hierarchical display is not as easy to decipher

as a two-way table.

Expression	Result

```
div4=If[Mod[#,4]==0,"div4","¬div4"]&;
ds24=Dataset@GroupBy[n7,{div2,div4}]
```

Out[•]=

odd	¬div4	{1, 3, 5, 7}
even	¬div4	{2, 6}
	div4	{4}

In this case, it is possible to produce a display comparable to the previous two-way table by providing a default value for every inner group and then updating it based on the data. For example, map an association holding default values across every row of the dataset. (The default values must come first.) Note that the result is no longer a true partition, because this adds an empty list.

Expression	Result

```
provideDefaults=<|"div4"→{},"¬div4"→{},#|>&;
provideDefaults/@ds24
```

Out[•]=

	div4	¬div4
odd	{}	{1, 3, 5, 7}
even	{4}	{2, 6}

To understand how this works, peer inside the dataset wrapper with the **Normal** command. This dataset is essentially an association whose values are also associations. Recall that applying **Map** to an association transforms the values, not the keys. For example, the key **"odd"** associates to the value **<|"¬div4"→{1,3,5,7}|>**. Applying **provideDefaults** to this value produces **<|"div4"→{},"¬div4"→{1,3,5,7}|>**. With this understanding, it is clear that mapping **provideDefaults** over the dataset **ds24** will provide these defaults wherever needed.

As of Mathematica version 12.1, control of dataset display remains tricky. It is affected by inferred type information, which is beyond the scope of this book. Sometimes the above trick fails to produce a two-way display rather than a hierarchical display. Applying **Transpose** twice to the dataset sometimes forces the two-way display. An alternative is to use a dataset for recoding, filtering, and aggregation, and then subsequently extract the results for use with **TableForm** or **Grid**.

Expression	Result

```
dta=Normal@Query[All,All,Length][provideDefaults/@ds24];
rowHeaders=Keys[dta];columnHeaders=Union@@Keys/@Values@dta;
values=Outer[dta,rowHeaders,columnHeaders];
TableForm[values,TableHeadings→{rowHeaders,columnHeaders}]
```

Out[•]=

	div4	¬div4
odd	0	4
even	1	2