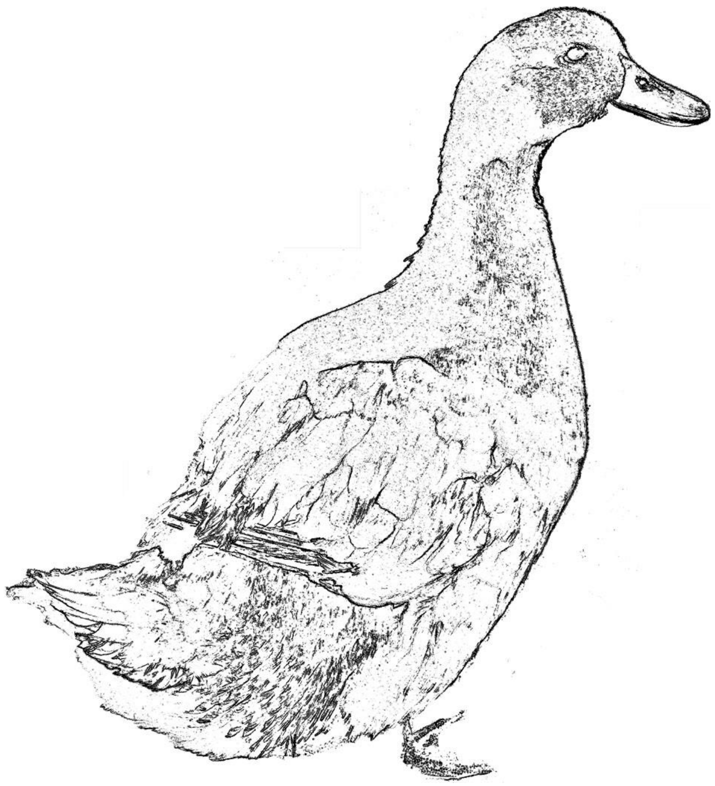
E-mail: easydatabase at gmail dot com

Perl Data::Table Cookbook

for Data::Table Version 1.77



Yingyao Zhou

Document Version 1.10
Updated on February 11, 2020

Table of Contents

1 Introduction 4
2 Input 7
2.1 Reading from a CSV file 7
2.2 Reading from a TSV file 8
2.3 Reading from a file 9
2.4 Reading from a database 9
2.5 Converting a memory string into a table object 10
2.6 Reading from a compressed data file 11
2.7 Reading/Writing an Excel file 11
3 Output 12
3.1 Writing a CSV file 12
3.2 Writing a TSV file 12
3.3 Displaying a table in HTML 13
3.4 Writing to an Excel file 15
4 Accessing Table 17
4.1 Getting a table cell 17
4.2 Getting table dimensions 18
4.3 Looping through table rows 18
4.4 Getting a table row 19
4.5 Getting a table column 19
4.6 Getting all column names 20
4.7 Modifying a table cell 20
4.8 Adding a new row 20
4.9 Deleting a row 21
4.10 Deleting rows 21
4.11 Adding a new column 22
4.12 Deleting a column 22
4.13 Moving a column 23
4.14 Swapping two columns 23

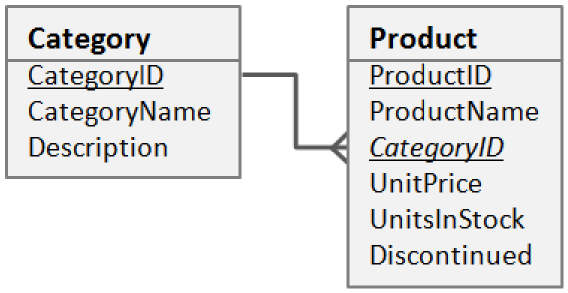
4.15 Replacing a column ... 23
4.16 Reorder columns ... 23
4.17 Renaming a column ... 24
5 Initializing Table ... 24
5.1 Initializing an empty table ... 24
5.2 Initializing a table by rows ... 25
5.3 Initializing a table by columns ... 25
5.4 Initializing a table by file handler or memory string ... 26
6 Table Filtering and Sorting ... 26
6.1 Filtering rows by array mode ... 26
6.2 Filtering rows by hash ... 27
6.3 Find rows by regular expression in any column ... 28
6.4 Filtering rows by a Boolean mask array ... 28
6.5 Getting a subset of a table ... 29
6.6 Sorting a table by multiple columns ... 29
7 Manipulating Two Tables ... 30
7.1 Joining two tables ... 30
7.2 Merging two tables row-wise ... 31
7.3 Merging two tables column-wise ... 32
8 Transforming a Table ... 33
8.1 Reshaping – melting and casting for table statistics ... 33
8.2 Grouping a table with aggregation functions ... 36
8.3 Pivoting a table ... 37
8.4 Looping through groups ... 39
9 Shortcuts ... 40
9.1 Useful constants ... 40
9.2 Sugar ... 40
10 Pitfalls ... 41
10.1 Internal data structures ... 41
10.2 Volatile methods ... 42

1 Introduction

The Table data structure is a two-dimensional spreadsheet, where each row represents a record and each column represents a property of the record. One can think of a Table as a matrix, where elements do not have to be numerical, however, all elements of a column share the same data type. This data structure is known as Table in relational database, as data.frame in R, DataTable in C#, and DataFrame in python. If Excel is one of the favorite applications our clients use, it probably indicates we, as analysts, have to deal with table data structure frequently. Table is such a universal data structure that it makes a great deal of sense to have it as a built-in data structure of any programming language, but unfortunately Perl misses it. The Data::Table module essentially fills this hole for Perl programmers. How do you currently read in a comma-separated value (CSV) file in to your Perl script? If you read it in line by line and split each line by comma, you can certainly benefit from using Data::Table module.

Effective use of Data::Table module enables us to focus on high-level data manipulations and leads to much cleaner code. This cookbook provides examples on how to use Data::Table to construct solutions for various computational problems, therefore, it serves as a reading material complementary to the reference-style CPAN document[1]. The author read "R Cookbook"[2] and liked it; therefore some topics in this book may follow the examples found there.

In this cookbook, we often use two tables obtained from the Microsoft Northwind sample dataset: "Category" and "Product".



Category consists of eight food categories, with CategoryID as its primary key. Product consists of 77 products, where ProductID is its primary key and CategoryID is a foreign key pointing to the Category table (see diagram above). Each category can have multiple products. The first five rows of the two tables are shown below in the comma-separated value (CSV) format:

```
CategoryID,CategoryName,Description
1,Beverages,"Soft drinks, coffees, teas, beers, and ales"
2,Condiments,"Sweet and savory sauces, relishes, spreads, and seasonings"
3,Confections,"Desserts, candies, and sweet breads"
4,Dairy Products,Cheeses
5,Grains/Cereals,"Breads, crackers, pasta, and cereal"
```

[1] http://search.cpan.org/perldoc?Data::Table

[2] Paul Teetor, R Cookbook, O'Reilly.

```
ProductID,ProductName,CategoryID,UnitPrice,UnitsInStock,Discontinued
1,Chai,1,18,39,FALSE
2,Chang,1,19,17,FALSE
3,Aniseed Syrup,2,10,13,FALSE
4,Chef Anton's Cajun Seasoning,2,22,53,FALSE
5,Chef Anton's Gumbo Mix,2,21.35,0,TRUE
```

First, let us get a taste of what Data::Table can do.

Problem

Assume we are interested in knowing the top three categories that have the largest total value of all in-stock active products.

Solution

```
use Data::Table;
# read all products into $t_product Data::Table object
my $t_product = Data::Table::fromFile("Product.csv");
# find all in-stock products
$t_product = $t_product->match_pattern_hash('$_{Discontinued} eq "FALSE"');
# calculate total cost of products in each category
my $t_category_cost = new Data::Table();
$t_product->each_group(['CategoryID'],
  sub {
    my ($t, $sum) = (shift, 0);
    map { $sum += $t->elm($_, 'UnitsInStock')*$t->elm($_, 'UnitPrice')} 0..$t->lastRow;
    $t_category_cost->addRow({CategoryID => $t->elm(0, 'CategoryID'), TotalCost => $sum});
  });
# read all categories into $t_category
my $t_category = Data::Table::fromFile("Category.csv");
# obtain a table of columns: CategoryID,CategoryName,Description,TotalCost
my $t = $t_category->join($t_category_cost, Data::Table::INNER_JOIN,
  ['CategoryID'], ['CategoryID']);
# get the three most costy categories
$t->sort('TotalCost', Data::Table::NUMBER, Data::Table::DESC);
$t = $t->subTable([0 .. 2]);
print $t->tsv(1);

# outputs
CategoryID      CategoryName    Description     TotalCost
8       Seafood Seaweed and fish        13010.35
1       Beverages       Soft drinks, coffees, teas, beers, and ales     12390.25
2       Condiments      Sweet and savory sauces, relishes, spreads      12023.55
```

Discussion

We need to read in all product data, filter out discontinued ones, calculate the `TotalCost` for each product category, append additional category annotation data (`CategoryName` and `Description`), keep the top three, and report.

`fromFile()` reads in a CSV/TSV file, it automatically detects the file format, including the presence of column header and OS-dependent line breaks. We then only keep the records where the Discontinued field equals `"FALSE"` using `match_pattern_hash()`. A new table $t_category_cost is constructed by calculating `TotalCost` for each product category using each_group(). We merge in additional category annotation data using join(), then sort() the resultant table and keep the top three categories with subTable(). The final data is reported in tab-delimited value (TSV) format.

Conventions

Perl code is printed with Consolas font. Screen output is printed with *Consolas Italic*.

When we run scripts in this book, always remember to include:

```
use Data::Table;
```

To save space, we always omit this line afterwards.

As we use two example tables, Category and Product, so often, we may also directly use $t_product and $t_category without initialization:

```
my $t_product = Data::Table::fromFile("Product.csv");
my $t_category = Data::Table::fromFile("Category.csv");
```

We may use variable $t without initialization:

```
my $t = new Data::Table();
```

$t is often used to represent a generic Data::Table object.

Our examples may skip the definition of an average routine, which is defined as the following.

```
sub average {
  my ($sum, $n) = (0, 0);
  map {$sum += $_; $n++} @_;
  return $sum/$n;
}
```

Contacts

I myself uses Data::Table module intensively, so this module is being actively maintained. Due to backward compatibility requirements, I am generally not able to redefine or refine method parameters, or rename methods. Otherwise, bug reports, feature requests, or usage questions can be sent to the email address: *easydatabase at gmail dot com*. People contribute to the improvement of the module may see their names appear in the Changes.txt file (in the CPAN package); please let me know if you do not want to be credited that way.

Borrowing the O'Reilly's tradition of showing an animal on the cover (this is NOT an O'Reily book, but I like its style), I picked Peking duck as the cover, not quite because of its taste, but rather because I had the fortune of growing some as a kid. It links to good old memories.

2 Input

2.1 Reading from a CSV file

Solution

```
my $t_product = Data::Table::fromCSV("Product.csv", 1);
```

Discussion

The first parameter is the file name, the second parameter indicates whether the first line of Product.csv is for column headers, here "1" stands for yes. If the file does not contain a header line, the columns will be automatically named as `col_1`, `col_2`, etc. We may also supply our own column name as the third parameter:

```
my $t_category = Data::Table::fromCSV("Category.csv", 1, ["ID", "Name", "Comment"]);
```

In the above example, the header line in the file will be ignored and the supplied column names are used instead.

The fourth optional parameter is a hash reference enabling further fine tuning, if not supplied, it defaults to:

```
{
  delimiter => ',',
  qualifier => '"',
  skip_lines => 0,
  OS => Data::Table::OS_UNIX,
  skip_pattern => undef,
  encoding => 'UTF-8'
}
```

The default delimiter and qualifier are suitable for CSV format. In some countries, `','` is used as the decimal point for numbers and `':'` as the delimiter. If this is the case, instead of overwriting the default every time one uses `fromCSV()`, one may choose to modify `%Data::Table::DEFAULTS` just once instead. Actually the default values of the fourth parameter come from the hash `%Data::Table::DEFAULTS`, which contains four entries:

```
{
  CSV_DELIMITER => ',',
  CSV_QUALIFIER => '"',
  OS => Data::Table::OS_UNIX,
  ENCODING => 'UTF-8' # since version 1.69
}
```

Reading a CSV could be much trickier, as line breaks is `"\n"` for UNIX, `"\r\n"` for PC, `"\r"` for MAC[3]. In addition, we may need to skip a few lines at the beginning of the file by using the

[3] http://en.wikipedia.org/wiki/Newline
"\r" is used up to Mac OS 9; "\n" is used since Mac OS X. Data::Table does not support other linebreak characters that maybe used by some infrequent operating systems.

`skip_lines` option. Or w might need to ignore all lines starting with `"#"`, by supplying a regular expression `'^#'` as the `skip_pattern`.

The method `fromCSV()` can also take a file handler instead of a file name, which may be useful for reading data from an already-opened file (see 2.5).

If we do not want to check ourselves whether a file has a header or was generated under which OS convention, simply use `fromFile()` described in 2.3.

See Also

See 2.3 for `fromFile()` as an alternative.

Method

`fromCSV(), %Data::Table::DEFAULTS`

2.2 Reading from a TSV file

Solution

```
        # Pretend Product.tsv is a tab-delimited file
        my $t_product = Data::Table::fromTSV("Product.tsv", 1);
```

Discussion

Reading from a TSV file is very similar to reading from a CSV file, except we cannot redefine delimiter (which is always `"\t"`) and qualifier (`""`, none). It supports the remaining options OS, skip_lines, and skip_pattern in its fourth parameter.

TSV supports a special additional option in the fourth parameter: `transform_element`, if set to true (1), it will convert an element `'\N'` into `undef`, `'\\'` into `"\\"`, `'\0'` into `"\0"`, `'\n'` into `"\n"`, `'\t'` into `"\t"`, `'\r'` into `"\r"`, `'\b'` into `"\b"`. Therefore, TSV would be a special format that can preserve element values such as `undef` or elements containing `"\n"` into a file and reads them back later, which cannot be handled by CSV format. E.g., a table element of empty string or `undef` are both presented as empty string in the CSV format, therefore both are treated as an empty string upon `fromCSV()`. There is no standard for TSV format; we here do what makes sense according to the ideas used for MySQL dump and import.

The method `fromTSV()` can also take a file handler instead of a file name, which may be useful for reading data from an already-opened file.

See Also

See 2.3 for `fromFile()` as an alternative.

Method

`fromTSV()`

2.3 Reading from a file

Solution

```
my $t_product = Data::Table::fromFile("Product.csv");
my $t_category = Data::Table::fromFile("Category.tsv");
```

Discussion

If we do not want to worry about whether the input file is TSV or CSV, whether the first line is a column header line or not, whether it is a PC format or a UNIX format, etc, just use `fromFile()`, which will employ certain heuristics in figuring out the right options. In case `fromFile()` fails, we may offer some hints to the method by providing a hash reference as the parameter:

```
{
  linesChecked => 2,
  OS => <hint>,
  has_header => <hint>,
  delimiter => <hint>,
  allowNumericHeader => 0
}
```

If the method gets a `<hint>` (e.g., OS => Data::Table::OS_PC), it will use it without trying to guess on its own. By default, `fromFile()` can guess the settings by peeking at the first two lines, which should be sufficient in most cases, however, we can ask it to check more lines by setting `linesChecked`. The method does open and close the file a few times, so theoretically it is a bit slower, however, practically, the penalty is negligible, therefore you are encouraged to use `fromFile()` over `fromCSV()` or `fromTSV()`, just let the machine do the labor.

Fields in CSV format could contain linebreaks, if the field is surrounded by qualifiers. `fromFile()` may have trouble in such case. Providing `{format => "CSV"}` can help. Since version 1.69, we `fromFile()` enable integers to be used as column names in `fromCSV()` and `fromTSV()`. However, since numeric column headers are rare, `fromFile()` assumes no numeric column header by default, when it tries to guess if the first line in file is a column header line or not. If numeric column header is intentionally allowed, we can hint `fromFile()` by specifying `{allowNumericHeader => 1}`.

Since version 1.69, we also allow user to specify an encoding method, which is defaults to $Data::Table::DEFAULTS{ENCODING}, set as UTF-8. E.g., to read a file encoded in cp1252:

```
my $t1 = Data::Table::fromFile("sample.cp1252.csv", {encoding=>'cp1252'});
```

Method

`fromFile()`

2.4 Reading from a database

Solution

```
# obtain a database handler first
my $dbh = DBI->connect("DBI:mysql:test", "test", "") or die $dbh->errstr;
# fromSQL() does the rest of the magic
my $t = Data::Table::fromSQL($dbh, "SELECT * FROM Product ORDER BY ProductID");
print $t->csv(1);
```

Discussion

Often we need to fetch records from a database; `fromSQL()` takes a database handler as the first parameter, and a SQL statement as the second. The SQL statement does not have to be a SELECT statement, if a statement such as INSERT/UPDATE/DELETE, the return Data::Table will simply be `undef`.

If the SQL expects parameter binding, `fromSQL()` takes an array reference as the third parameter.

```
my $t = Data::Table::fromSQL(
  $dbh,
  "SELECT * FROM Product WHERE ProductName like ? AND Discontinued=?",
  ['BOSTON%', 'FALSE']
);
```

In MySQL, a primary key of a record may often be set to AUTO_INCREMENT, i.e., database will assign the key for us. In this case, right after we INSERT a new record, we may use `LAST_INSERT_ID()` to obtain the value of the newly assigned key.

```
my $t = Data::Table::fromSQL($dbh, "SELECT LAST_INSERT_ID()");
print $t->elm(0, 0);
```

Instead of passing a SQL string, we can also use a DBI::st object. This might be desirable if we have to repeat a SQL statement many times:

```
my $sth = $dbh->prepare("SELECT * FROM Product");
# since the first parameter below is ignored, $sth already contains connection info
# so we can use undef instead.
my $t = Data::Table::fromSQL($dbh, $sth);
```

Notice `fromSQL()` reads in the whole table from database all at once, which makes it really convenient. A hint, if we reads large amount of data across a slow network, read data in bulk might boost performance, try to set `$dbh->{RowCacheSize}=16*1024` (this is really a topic belongs to the DBI module).

Method

`fromSQL()`

2.5 Converting a memory string into a table object

Solution

```
my $s ="a,b,c\n1,2,3";
# treat the string as if it is a file
open my $fh, "<", \$s or die "Cannot open in-memory file\n";
```

```
my $t= Data::Table::fromCSV($fh, 1);
print $t->csv(1);

a,b,c
1,2,3
```

Discussion

`fromCSV()` and `fromTSV()` can takes a file handler instead of a file name. However, `fromFile()` cannot take a file handler, because `fromFile()` needs to peek into a file in order to figure out the right parameters, then closes and reopens the file with `fromCSV()` and `fromTSV()`.

Method

`fromCSV(), CSV()`

2.6 Reading from a compressed data file

Solution

```
my $t=Data::Table::fromCSV("zcat Product.csv.gz |", 1);
```

Discussion

Making use of the UNIX pipe, one can read from a file that is the result of preprocessing commands.

Method

`fromCSV()`

2.7 Reading/Writing an Excel file

Solution

```
use Data::Table::Excel qw(tables2xls tables2xlsx xls2tables xlsx2tables);

# write two tables into a Northwind.xls file, using Excel 2003 binary format
tables2xls("Northwind.xls", [$t_category, $t_product], ["Category","Product"]);

# read tables from the Northwind.xls file
my ($tableObjects, $tableNames)=xls2tables("Northwind.xls");
$t_category = ($tableName[0] eq "Category")? $tableObjects[0]:$tableObjects[1];
$t_product = ($tableName[0] eq "Product")? $tableObjects[0]:$tableObjects[1];

# or read tables from the Northwind.xlsx file (Excel 2007 format)
my ($tableObjects, $tableNames)=xlsx2tables("Northwind.xlsx");

for (my $i=0; $i<@$tableNames; $i++) {
  print "*** ". $tableNames->[$i], " ***\n";
  print $tableObjects->[$i]->csv;
}

Outputs:
*** Category ***
```

```
CategoryID,CategoryName,Description
1,Beverages,"Soft drinks, coffees, teas, beers, and ales"
2,Condiments,"Sweet and savory sauces, relishes, spreads, and seasonings"
3,Confections,"Desserts, candies, and sweet breads"
...

*** Product ***
ProductID,ProductName,CategoryID,UnitPrice,UnitsInStock,Discontinued
1,Chai,1,18,39,FALSE
2,Chang,1,19,17,FALSE
3,Aniseed Syrup,2,10,13,FALSE
...
```

Discussion

We release `Data::Table::Excel` module into CPAN, which enables one to read/write Excel files (both 2003 and 2007 formats). Since this module depends on other Perl modules, including `Spreadsheet::WriteExcel`, `Spreadsheet::ParseExcel`, `Spreadsheet::XLSX`, and `Excel::Writer::XLSX`, we decided not to add these methods as part of the `Data::Table` module, in order not to break codes for existing `Data::Table` users. There are more options for controlling Excel files generation, see 3.4.

Method

`xls2tables(), xlsx2tables()`

3 Output

3.1 Writing a CSV file

Solution

```
my $t_product = $t->fromFile("Product.tsv");
$t_product->csv(1, { file => "Product.csv" });
```

Discussion

By default, `csv()` method serializes the table into a string, if file name is not supplied. The first parameter defines whether column header line should be exported. The second parameter is a hash reference enabling us to define file name, line break, delimiter, and qualifier. E.g.,

```
$t_product->csv(1, {OS => Data::Table::OS_PC, delimiter => "\t", qualifier => "" });
```

This basically returns the data in a TSV-formatted string for Windows PC.

3.2 Writing a TSV file

Solution

```
my $t_product = $t->fromFile("Product.csv");
$t_product->tsv(1, { file => "Product.tsv" });
```

Discussion

This method is very similar to `csv()`method, except it has fixed delimiter and qualifier. One can switch off `transform_element` by passing `{transform_element => 0}` in the second parameter, if so desire. As mentioned in `fromTSV()` (see 2.2) method, TSV format can distinguish `undef` from emtpy string, because `undef` is stored as `'\N'` in the file according to MySQL convention (`transform_element` is on by default). The cell value can be restored to unde, when the file is read back with `fromTSV()`. If CSV`()` is used instead, `undef` will become an empty string. `TSV()` and `fromTSV()` is most suitable to serialize and deserialize a table containing special characters.

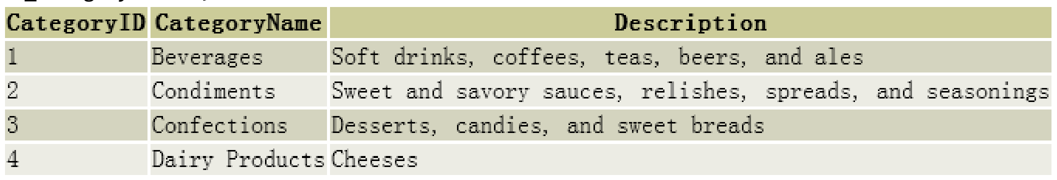
Method

`tsv()`

3.3 Displaying a table in HTML

Solution

```
$t_category->html;
```

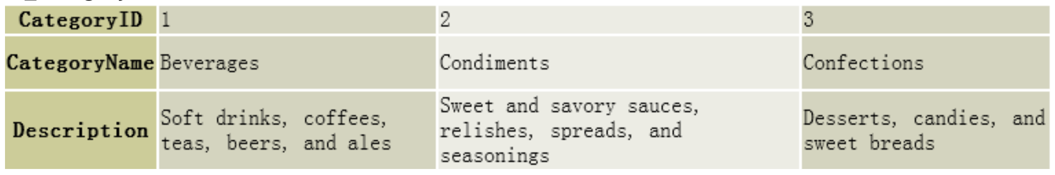


CategoryID	CategoryName	Description
1	Beverages	Soft drinks, coffees, teas, beers, and ales
2	Condiments	Sweet and savory sauces, relishes, spreads, and seasonings
3	Confections	Desserts, candies, and sweet breads
4	Dairy Products	Cheeses

Discussion

If we want to show the table in wide mode, i.e., each row is displayed as a column (typically for a fat table of many columns but few rows), use

```
$t_category->html2;
```



CategoryID	1	2	3
CategoryName	Beverages	Condiments	Confections
Description	Soft drinks, coffees, teas, beers, and ales	Sweet and savory sauces, relishes, spreads, and seasonings	Desserts, candies, and sweet breads

By default, the table displayed uses a default color theme, where odd and even rows have alternative-color background and header has a darker-color background. We can fine tune the HTML table by providing additional parameters.

The first parameter can be a color array reference, default to `["#D4D4BF","#ECECE4","#CCCC99"]`, specifying colors for odd rows, even rows, and column header, respectively (default colors are shown in the screenshots above). As CSS now becomes so popular, one would probably choose to specify a hash reference that defines CSS classes for odd, even, and header tags, by default the class names are:

```
{even => "data_table_even", odd => "data_table_odd", header => "data_table_header"}
```

The method takes additional parameters defining properties for tags including <TABLE>, <TR>, <TH>, and <TD> in the form of hash reference.

```
$t_category->html(
  {even => "my_even", odd => "my_odd", header => "my_header"},
  # properties for <TABLE> tag
  {border => '1'},
  # properties for <TR> tag
  {align => 'left'},
  # properties for <TH> tag
  {align => 'center'},
  # properties for <TD> tag
  {
    CategoryName => 'align="right" valign="bottom"',
    2 => 'align="left"'
  });

# outputs HTML code
<table border="1">
<thead>
<tr align="left" class="my_header"><th align="center">CategoryID</th><th
align="center">CategoryName</th><th align="center">Description</th></tr>
</thead>
<tbody>
<tr align="left" class="my_odd"><td>1</td><td align="right"
valign="bottom">Beverages</td><td align="left">Soft drinks, coffees, teas, beers, and
ales</td></tr>
<tr align="left" class="my_even"><td>2</td><td align="right"
valign="bottom">Condiments</td><td align="left">Sweet and savory sauces, relishes, spreads,
and seasonings</td></tr>
...
</tbody>
</table>
```

In this example, it adds CSS classes: `my_header`, `my_oldd`, `my_even` to table header row, odd rows, and even rows, respectively. The actually colors for these classes are typically defined in .css files included somewhere else in the HTML page. It generates `<table border="1">`, because we provide `{border => "1"}` as the parameter and one can certainly specify more name-value pairs in this hash structure to further control `<table>`. `<tr algin="left">`, `<th align="center">` are the results of corresponding parameters. The pattern here is: each hash key becomes the name of the tag attributes and hash value becomes the attribute value. `<td>` controlling parameter takes the format that the keys can be either column names or column indices (the column index is the numerical position of a column, e.g., the first column has an index of 0, the value is the second column has an index of 1, etc), and the hash values are the tag attributes go into the corresponding `<td>` tag. In our example above, we add `'align="right" valign="bottom"'` to each `<td>` tag corresponding to the `CategoryName` column, and add `'align="left"'` to column 2 (the `Description` column, the third column in the table, with a column index of 2).

Nowadays, one almost always wants to provide a `{class => "classname"}` and then define class properties in .css files. So instead of `CategoryName => 'align="right"` `valign="bottom"'`, it almost makes more sense to write `CategoryName => 'class="myCategoryName"'`, and define `myCategoryName` within a .css file.

Since version 1.74, users can control the tags for each table cell, including header cells. The way to accomplish this is by providing a callback subroutine. The callback subroutine can take the following arguments: (tag_hash, row_index, col_index, col_name, table_ref). The tag_hash is the current hash of tags for the given cell, row_index, col_index and dataframe allows us to determine which cell is the target cell, col_name is provided for convenience. Notice, if row_index is -1, it indicates the target cell is a column header cell. The following example which highlight the cells for expensive products in orange and cheap ones in blue, color the cells for discontinued item in gray and active products in purple.

```
$t=Data::Table::fromCSV("Data-Table-1.74/Product.csv",1,undef, {'OS'=>1});
$t = $t->subTable([0..5]);

my $callback = sub {
  my ($tag, $row, $col, $colName, $table) = @_;
  if ($row >=0 && $colName eq 'UnitPrice') {
    $tag->{'style'} = 'background-color:'. (($table->elm($row, $col)>=20) ?
'#fc8d59':'#91bfdb') . ';';
  }
  if ($row >=0 && $colName eq 'Discontinued') {
    $tag->{'style'} = 'background-color:'. (($table->elm($row, $col) eq 'TRUE') ?
'#999999':'#af8dc3') .';';
  }
  return $tag;
};

print $t->html(undef, undef, undef, undef, undef, undef, $callback);
```

ProductID	ProductName	CategoryID	UnitPrice	UnitsInStock	Discontinued
1	Chai	1	18	39	FALSE
2	Chang	1	19	17	FALSE
3	Aniseed Syrup	2	10	13	FALSE
4	Chef Anton's Cajun Seasoning	2	22	53	FALSE
5	Chef Anton's Gumbo Mix	2	21.35	0	TRUE
6	Grandma's Boysenberry Spread	2	25	120	FALSE

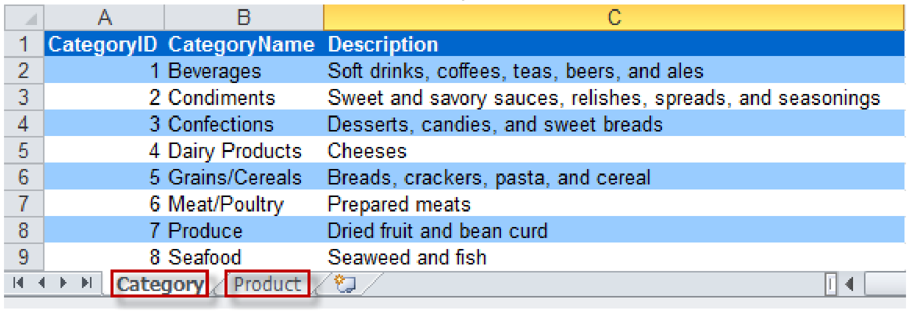
Method

`html(), html2()`

3.4 Writing to an Excel file

Solution

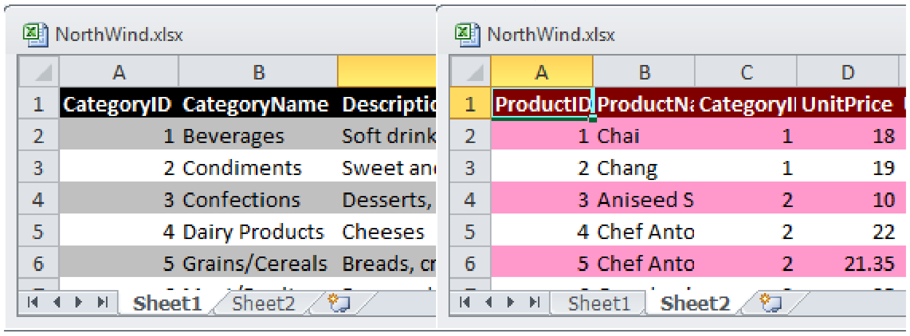
```
# write two tables into a Northwind.xls file, using Excel 2003 binary format
tables2xls("Northwind.xls", [$t_category, $t_product], ["Category","Product"]);
# the workbook will contain two sheets, named Category and Product
# parameters: output file name, an array of tables to write, and their corresponding names
```



	A	B	C
1	CategoryID	CategoryName	Description
2	1	Beverages	Soft drinks, coffees, teas, beers, and ales
3	2	Condiments	Sweet and savory sauces, relishes, spreads, and seasonings
4	3	Confections	Desserts, candies, and sweet breads
5	4	Dairy Products	Cheeses
6	5	Grains/Cereals	Breads, crackers, pasta, and cereal
7	6	Meat/Poultry	Prepared meats
8	7	Produce	Dried fruit and bean curd
9	8	Seafood	Seaweed and fish

Category Product

```
# write two tables into a Northwind.xls file, using Excel 2007 compressed XML format
tables2xlsx("NorthWind.xlsx", [$t_category, $t_product], undef,
[['silver','white','black'], [45,'white',37]]);
```



NorthWind.xlsx

	A	B	
1	CategoryID	CategoryName	Descriptic
2	1	Beverages	Soft drink
3	2	Condiments	Sweet an
4	3	Confections	Desserts,
5	4	Dairy Products	Cheeses
6	5	Grains/Cereals	Breads, cr

Sheet1 Sheet2

NorthWind.xlsx

	A	B	C	D
1	ProductID	ProductN:	CategoryI	UnitPrice
2	1	Chai	1	18
3	2	Chang	1	19
4	3	Aniseed S	2	10
5	4	Chef Anto	2	22
6	5	Chef Anto	2	21.35

Sheet1 Sheet2

Discussion

See 2.7 for how to read tables from an Excel file.

In the first example, we provide the table names, so that they become the names used in the resultant spread sheets. In the second example, we skip the table names, so that the two sheets are named “Sheet1” and “Sheet2”. We provide two different color arrays, one for each table. Each color array must contain three color elements, controlling colors used for odd rows, even rows and the header row. If colors are not found for a table, it will inherit colors defined for the previous table. The color index is defined by the docs\palette.html file as part of the CPAN Spreadsheet::WriteExcel package distribution.

The method can actually take a fifth parameter. Say if we use [1, 0] in the above example, the first Category table will be displayed in portrait mode (each row is a category record, which is the default), the second Product table will instead be displayed in landscape mode (each column is a product, which is similar to the effect of the `html2()` method).

Method

`tables2xls(), tables2xlsx(),`

4 Accessing Table

4.1 Getting a table cell

Solution

```
$t_product->elm(0, "ProductID");
# returns 1
```

Discussion

The coordinate of a cell is defined by its row number and column index. Column index can be the column name or the column number. Columns are numbered as 0, 1, 2, ..., etc. If the `ProductID` column is the first column, the above statement is identical to

```
$t_product->elm(0, 0);
# returns 1
```

We encourage you to use column name instead of column index whenever possible, as it is more readable and stable (when one add or delete other columns, column number changes but column name stays the same). Before version 1.69 there is a constraint on the column name, i.e., it cannot be a numerical value, because it will be very confusing. Since version 1.69, we relax this requirement. This is to facilitate the circumstance, where we are given a data file with numeric column names. However, if you create your own Data::Table object, we strongly recommend you not to use integer column names to avoid any confusion. Users should not worry about any performance penalty in using column names; each `Data::Table` object contains a lookup table, so that column name to column number conversion is done with a single hash lookup. Sometimes we need the column number, use

```
$t_product->colIndex("ProductID");
# returns 0
```

A return value of -1 by `colIndex()` means the column name does not exist. One can use a shortcut method `hasCol()` instead:

```
$t_product->hasCol("Non-Exist-Column");
# returns false
```

Since 1.69, we allow integers to be used as column header. In calling `colIndex(123)`, 123 is first interpreted as a column name string, if not found, it is interpreted as the numeric column index. This could certainly create undesirable side effects, therefore, be careful when integer column names are used. If “123” is a column name and you intentionally would like to access the 124th column, you can access by its column name, i.e., $t->col($t->header)[123]).

Method

`elm(), colIndex(), hasCol()`

4.2 Getting table dimensions

Solution

```
$t_product->nofRow;
# returns 77
$t_product->nofCol;
# returns 6
$t_product->isEmpty;
# returns false
$t_product->lastRow;
# returns 76
$t_product->lastCol;
# returns 5
```

Discussion

`lastRow()` and `lastCol()` are just syntax sugars that makes looping easier to read.

Method

`nofRow(), nofCol(), isEmpty(), lastRow(), lastCol()`

4.3 Looping through table rows

Solution

To loop through all rows in a table, feel free to choose any one of the following styles:

```
for (my $i = 0; $i < $t_product->nofRow; $i++) { ... }
foreach my $i (0 .. $t_product->nofRow - 1) { ... }
foreach my $i (0 .. $t_product->lastRow) { ... }
map { ... } 0 .. $t_product->nofRow – 1;
map { ... } 0 .. $t_product->lastRow;
```

We can also use `iterator()` to loop through a table.

```
my $next = $t_product->iterator();
while (my $row = &$next) {
  # access the row element using hash
  print 'Processing Row: '.&$next(1). "\n";
  print 'ProductName: '.$row->{ProductName}. "\n";
}

Processing Row: 0
ProductName: Chai
Processing Row: 1
ProductName: Chang
Processing Row: 2
ProductName: Aniseed Syrup
...
```

`iterator()` returns a enumerator, which is a reference to a subroutine. Calling `$next->()` (or `&$next`) will fetch the next table row as hash reference, so that one can easily access elements with. If we want to iterator the table backward, i.e., from the last row to the first, use `iterator({reverse => 1})`. Notice that modifying the value in the hash `$row-`

`>{ProductName} = "New Name"` does not really change the values in the table, as the hash only contains a copy of the table elements. To modify the original table, we need to know the row index of the record we just fetched, for that, use `&$next(1)` to obtain the row index. E.g., one can update the original table with:

```
        $t_product->setElm(&$next(1), 'Product Name', 'New Name');
```

When no more element can be returned, `&$next` returns `undef`.

Method

`iterator(), setElm()`

4.4 Getting a table row

Solution

```
        $t_product->row(0);
        # returns the first row as an array (not an array reference!)
        1, Chai, 1, 18, 39, FALSE

        $t_product->rowHashRef(0);
        # returns the first row as a reference to a hash (not a hash!)
        {
          ProductID => 1,
          ProductName => "Chai",
          CategoryID => 1,
          UnitPrice => 18,
          UnitsInStock => 39,
          Discontinued => "FALSE"
        }
```

Discussion

The `row()` method returns an array, one then needs to access elements by array index. The `rowHashRef()` method returns a hash reference, which is often more convenient as one can access elements by their names. Be aware that the elements are a copy, therefore, modifying elements in the returned data structure does not modify the cells in the original table object.

Method

`row(), rowHashRef()`

4.5 Getting a table column

Solution

```
        my @product_names = $t_product->col("ProductName");
        # get descriptions from the third column
        my @descriptions = $t_product->col(2);
```

Discussion

Wherever a column index is expected as a parameter in any `Data::Table` method, one can always use column name (recommended) instead.

Method

`col()`

4.6 Getting all column names

Solution

```
my @column_names = $t_product->header;
```

4.7 Modifying a table cell

Solution

```
$t_product->setElm(0, "ProductName", "New Product Name for ProductID 1");
```

Discussion

The first two parameters specify the row and column coordinate of the cell, the third parameter is the new value to be assigned to the cell.

`setElm()` can actually modify multiple cells at once, i.e., the first and second parameter can also be an array reference, that defines a grid of cells to change. The example below sets the `UnitPrice` of the first three products to 20. We can specify multiple columns as well.

```
$t_product->setElm([0..2], 'UnitPrice', 20);
```

Method

`setElm()`

4.8 Adding a new row

Solution

```
$t_product->addRow([78, "Extra Tender Tofu", 7, 23.25, 20, "FALSE"]);
$t_product->addRow(
{
  ProductID => 79, ProductName => "Extra Firm Tofu",
  CategoryID => 7, UnitPrice => 23.25,
  UnitInStock => 20, Discontinued => "FALSE"
});
```

Discussion

If one provides a row in the format of an array reference, the order of values has to exactly match the column order and the size of the array has to be the same as `nofCol()`. If one provides a hash reference, elements are matched by their keys. An element will be defaults to `undef`, if that field is not found in the hash.

By default, the new row is appended as the last row. However, one can also specify the location of the new row. The following statement would insert the new row as the first row (index 0). Appending a row to the end is faster than inserting a row elsewhere, so just leave the third element unspecified, if we do not care about the location of the new row.

```
$t_product->addRow([78, "Extra Tender Tofu", 7, 23.25, 20, "FALSE"], 0);
```

By default, the row hash the new row is appended as the last row. However, one can also specify the location of the new row. The following statement would insert the new row as the first row (index 0). Appending a row to the end is faster than inserting a row elsewhere, so just leave the third element unspecified, if we do not care about the location of the new row.

It is an error, if the row array contains more elements than `nofCol`. A row hash element is ignored, if it is not an existing column name. Occassionally, we might want to automatically create new columns, so that extra supplied elements are kept. Use an `{addNewCol => 1}` as the third parameter.

```
# automatically adds a new column "col7".  All existing rows has undef as the value.
$t_product->addRow([78, "Extra Tender Tofu", 7, 23.25, 20, "FALSE", "Y"], undef,
  {addNewCol => 1});

# automatically creates a new column "Comment"
$t_product->addRow(
{
  ProductID => 79, ProductName => "Extra Firm Tofu",
  CategoryID => 7, UnitPrice => 23.25,
  UnitInStock => 20, Discontinued => "FALSE",
  Comment => "Softer than rocks"
}, undef, {addNewCol => 1});
```

Method

`addRow()`

4.9 Deleting a row

Solution

```
#delete the last row
my $row = $t_product->delRow($t_product->nofRow - 1);
```

Discussion

It returns the deleted row as an array reference; most often we will ignore the returned value.

Method

`delRow()`

4.10 Deleting rows

Solution

```
        my @row = $t_product->delRows([0 .. 2]); #delete the first three rows
```

Discussion

It returns an array containing references to the deleted rows, feel free to ignore.

Method

`delRows()`

4.11 Adding a new column

Solution

```
        $t_category->addCol(["BV","CD","CF","DR","GC","MP","PR","SF"], "Code");
        $t_category->addCol("No comment yet", "Comment");
```

Discussion

A column is provided as an array reference, the size of the array must match `nofRow`. The second parameter is the column name. By default the new column is appended to the end, unless one specifies a column index number as the third parameter. To add a new column as the new first column:

```
        $t_category->addCol(["BV","CD","CF","DR","GC","MP","PR","SF"], "Code", 0);
```

Sometimes it is convenient to create a new column with a default value; we just pass the default value as the first parameter. The second example in Solution creates a new column “Comment” and initialize all elements with “No comment yet”. To initialize the element with `undef`, just use `undef` as the first parameter.

Method

`addCol()`

4.12 Deleting a column

Solution

```
        my @discontinued = $t_product->delCol("Discontinued");
        my ($uis, $disc) = $t_product->delCols(["UnitInStock","Discontinued"]);
```

Discussion

`delCol()` returns a reference to the deleted column array. `delCols()` returns and array containing references to the deleted columns. One typically ignores the returned values.

Method

`delCol(), delCols()`

4.13 Moving a column

Solution

```
        # move ProductName to the first column
        $t_product->moveCol("ProductName", 0);
```

Discussion

The second parameter specified the new position that the column will be moved into. The method can take a third optional parameter, which can be used to rename the column.

```
      $t_product->moveCol("ProductName", 0, "NewProductName");
```

Method

`moveCol()`

4.14 Swapping two columns

Solution

```
      # swap the locations of the two columns specified
      $t_product->swap("ProductID", "ProductName");
```

4.15 Replacing a column

Solution

```
        # create a new column array, converting TRUE/FALSE into 1/0.
        my @discontinued = ();
        map { push @discontinued, $_ eq "TRUE" ? 1:0 } $t_product->col("Discontinued");
        # replace the original column with the new numerical column
        $t_product->replace("Discontinued", \@discontinued);
```

Discussion

One can also provide a third parameter, specified a new column name.

Method

`replace()`

4.16 Reorder columns

Solution

```
        # rearrange all columns based on their alphanumerical order
        my @new_order = sort {$a cmp $b} $t_product->header;
        $t_product->reorder(\@new_order);
```

Discussion

`reorder()` is used to sort table columns into a particular order. For existing columns that are not in the specified new column array, they are moved to the end of the table. However, if one intends to delete the columns that are not specified, use a second parameter `{keepRest => 0}`.

This method is sort of similar to `subTable(undef, \@new_order)` as described later. However, `subTable()` makes a copy and returns a new table, while `reorder()` modifies the current table. Sometimes, `reorder()` could be much faster than `subTable()`, when an internal table copy is avoided.

Method

`reorder()`

4.17 Renaming a column

Solution

```
    $t_product->rename("ProductName", "Product_Name");
```

Method

`Rename()`

5 Initializing Table

5.1 Initializing an empty table

Solution

```
        # an empty table with no column, therefore must contain no row
        $t = new Data::Table();
        # an empty table with two columns: ID and Name, but has no row yet
        $t = new Data::Table([], ["ID", "Name"]);
```

Discussion

To create an empty table,

```
        $t = new Data::Table();
```

is equivalent to

```
        $t = new Data::Table([], [], Data::Table::ROW_BASED)
```

The first parameter is data matrix, the second parameter is a reference to the column header array, and the third parameter specifies whether the data matrix is ROW_BASED or COL_BASED, which does not really matter for an empty table.

One might need an empty table for coding elegance. Say if one would like to row merge five CSV files t1.csv, t2.csv, …, t5.csv into one table:
Instead of:

```
$t = Data::Table::fromFile("t1.csv");
foreach my $i (2..5) {
  $t->rowMerge(Data::Table::fromFile("t$i.csv");
}
```

One can write:

```
$t = new Data::Table();
foreach my $i (1..5) {
  $t->rowMerge(Data::Table::fromFile("t$i.csv");
}
```

Method

new

5.2 Initializing a table by rows

Solution

```
$t = new Data::Table(
  [[1, "Chai"], [2, "Chang"], [3, "Aniseed Syrup"]],
  ["ProductID", "ProductName"],
  Data::Table::ROW_BASED
);
```

Discussion

The method requires a data matrix as the first parameter, which is a reference to an array of row reference, followed by a column name array reference as the second parameter, then by a constant `Data::Table::ROW_BASED`. ROW_BASED indicates the table data matrix is organized by rows.

Method

new, `Data::Table::ROW_BASED`

See Also

5.3

5.3 Initializing a table by columns

Solution

```
$t = new Data::Table(
  [[1,2,3], ["Chai","Chang","Aniseed Syrup"]],
  ["ProductID", "ProductName"],
  Data::Table::COL_BASED
);
```

Discussion

This is similar to the previous example, the difference is the matrix data are given in columns, therefore we use constant `Data::Table::COL_BASED`.

Method

`Data::Table::COL_BASED`

See Also

5.2

Method

`new`

5.4 Initializing a table by file handler or memory string

See Also

Read 2.5

6 Table Filtering and Sorting

6.1 Filtering rows by array mode

Solution

```
my $price_index = $t_product->colIndex("UnitPrice");
my $status_index = $t_product->colIndex("Discontinued");
$t_expensive = $t_product->match_pattern(
  '$_->[' .$price_index. '] >20 && $_->[' .$status_index. '] eq "FALSE"');
```

Discussion

Method `match_pattern()` takes a string, which represents the filtering condition. The filtering string is evaluated against each table row, where the table row is passed in as `$_`, a reference to the row array. The filtering expression leads to a true/false result, which indicates whether the table row is skipped or returned. Since the expression only have access to an array reference, it cannot access row elements by name, but can only access by its column index. Therefore, we have to first obtain the column index in `$price_index` and `$status_index` and use them to construct the filtering expression. `$_->[$price_index]` refers to the `UnitPrice` value of the row passed in. Notice we use single quote to enclose `'$_'`, so that the dollar sign can be preserved and passed into the method.

This syntax is a bit awkward and makes this method hard to use. Method `match_pattern_hash()` is an optimized version that is much easier to use (see 6.2).

If the second parameter is set to TRUE, it is in a counting mode, i.e., returns the number of rows that matches, otherwise, it returns a table containing all the matched rows.

Method

`match_pattern(), colIndex(), match_string(), match_pattern_hash()`

6.2 Filtering rows by hash

Solution

```
my $t_expensive = $t_product->match_pattern_hash(
   '$_{UnitPrice}>20 && $_{Discontinued} eq "FALSE"');
print "Expensive/in-stock products found: ".$t_expensive->nofRow."\n";
print "Cheap/discontinued products found: ".
       scalar (grep /0/, @{$t_product->{OK}}) ."\n";
```

Discussion

Method `match_pattern_hash()` is probably more convenient than `match_pattern()`, because the pattern is applied to a row hash `%_` instead of an array reference `$_`. This means, we can refer to elements in a row by their names, instead of having to figuring out their column indices first. By using column names, the filtering expression is not only more readable, it also remains unchanged even other columns are added to or deleted from the table. Therefore, users should choose `match_pattern_hash()` over `match_pattern()`.

If the second parameter is set to TURE, it is in a counting mode, i.e., returns the number of rows that matches, otherwise, it returns a table containing all the matched rows.

All three filtering methods: `match_pattern_hash()`, `match_pattern()` and `match_string()` store the match results in an internal array call @$OK. After match method call, `@{$t_product->{OK}}` contains a Boolean array (0,0,0,1,0, …) , where `'1'` for the rows that evaluated to be true and `'0'` for false rows. For that reason, to count cheap/discontinued products, we just count the number of elements of value 0 using:

```
grep /0/, @{$t_product->{OK}}
```

The OK array can be used as a parameter in `rowMask()` method (see 6.5) to obtain the rows that do not match the pattern.

All three filtering methods: `match_pattern_hash()`, `match_pattern()` and `match_string()` also stores the match row indices in an internal array call @$MATCH. This is quite convenient, if we need to modify the matched rows:

```
# match returns all matched row ids in $t_product->{MATCH} (ref to row ID array)
$t_product->match_pattern_hash('$_{UnitPrice} > 20');
# create a new column, with 'No' as the default value
$t_product->addCol('No', 'IsExpensive');
# use $t_product->{MATCH} to set values for multiple Elements
$t_product->setElm($t_product->{MATCH}, 'IsExpensive', 'Yes');
```

In the above example, we find all the rows that have UnitPrice > 20. Then use the `$t->`{MATCH} array reference to flag those rows as expensive.

Method

`match_string(), match_pattern_hash(), match_pattern(), setElm()`

6.3 Find rows by regular expression in any column

Solution

```
print $t_product->match_string('tofu', 1, 1);
# output 2
```

Discussion

Method `match_string()` takes a regular expression and search all cells (all rows and all columns). As long as there is a column in a row matches the pattern, the row is considered a match. The second parameter indicatea whether case is ignored during the regular expression match. If the third parameter is set to TRUE, it is in a counting mode, i.e., returns the number of rows that matches, otherwise, it returns a table containing all the matched rows. This method is not used as often as `match_pattern_hash()`, but meant to be by those who are either "lazy" or find `match_pattern()` too hard to use.

Method

`match_string(), match_pattern_hash(), match_pattern()`

6.4 Filtering rows by a Boolean mask array

Solution

```
$t_expensive = $t_product->match_pattern_hash(
   '$_{UnitPrice}>20 && $_{Discontinued} eq "FALSE"');
$t_cheapORdiscontinued = $t_product->rowMask($t_product->{OK}, 1);
```

Discussion

As discussed previously, `$t_proudct->{OK}` is a Boolean array reference, where each element is 1 or 0 for rows that matches or not matches. This array is called a "mask". `rowMask($maskArrayRef)` would return a table containing all Okay rows. If the second parameter (`complement`) is set to true, it returns the rows that are NOT okay.

When the string argument passed to the method is evaluated internally, the variable scope outside the method is invisible. For instance, if we have $val = 5, an argument `'$_{UnitPrice}> $val'` is invalid, as the evaluation inside cannot see $val. The workaround is to evaluate the variable before feed it to the method. In this case, use `'$_{UnitPrice}> '.$val` instead.

Method

`rowMask(), match_pattern_hash()`

6.5 Getting a subset of a table

Solution

```
$t = $t_product->subTable([0..2], ["ProductID", "ProductName"]);
```

Discussion

This method copies a rectangle region of the table by specifying the row numbers and the columns. The data are copied and a new table is returned. If all rows are to be kept, use `undef` for the row indices. If all columns are to be kept, use `undef` for the second parameter.

The method `subTable()` can also select rows by a Boolean row mask array reference as the first parameter, however, in that case, we should provide a third parameter `{useRowMask => 1}`, so that the method knows not to look for row indices in the first parameter (see 6.2, 6.4 for row mask). The row mask array typically is the output of the filtering methods, such as `match_pattern_hash()`, `match_pattern()`, or `match_string()`. For example, to calculate the average price of products in category 5:

```
$t_product->match_pattern_hash('$_{CategoryID} == 5', 1);
average($t_product->subTable($t_product->{OK}, ['UnitPrice'], {useRowMask=>1})->col(0));
```

This certainly could also be accomplished with the code below:

```
average($t_product->match_pattern_hash('$_{CategoryID} == 5')->col('UnitPrice'));
```

The second approach returns all the table columns first, then only keeps the UnitPrice column. The first approach might have a small performance advantage, when it comes to a fairly large-scale data set. So the example here is just for the sake of illustrating the `useRowMask` mode.

Method

`subTable(), match_pattern_hash(), match_pattern(), match_string()`

6.6 Sorting a table by multiple columns

Solution

```
$t_product->sort('Discontinued', Data::Table::STRING, Data::Table::DESC,
                 'UnitPrice', Data::Table::NUMBER, Data::Table::ASC);
```

Discussion

For each column used in the sorting, one should specify its name (or certainly column index), data type (STRING or NUMBER), and sort direction (ASCending or DESCending). The method can take any number of sorting columns. Notice that the method reorders the rows in the original table; it does not return a new table, as this is most likely the desirable behavior.

Sometimes it is necessary for `sort()` to use a user supplied comparator, when neither numerical nor alphabetic ordering is correct. A custom comparator can be supplied where data type parameter is normally expected. Here is an example, where we would like to sort biological plate data by its well ID. A plate is a two dimensional matrix of wells, where each well contains some biological sample. The well identifier has the format of starting with an alphabetic character representing the row (A, B, C, …), followed by an underscore, then followed by a numeric number representing the column, this is very similar to a cell reference in Excel. In order to sort well IDs correctly, one would first string compare the first character, if tied, compare the second numeric part.

```
# create a table of one column PlateWell
$Well=["A_1", "A_2", "A_11", "A_12", "B_1", "B_2", "B_11", "B_12"];
$t = new Data::Table([$Well], ["PlateWell"], 1);
# if we sort the table by string or by number, the results are not what we want
$t->sort("PlateWell", Data::Table::STRING, Data::Table::ASC);
print join(" ", $t->col("PlateWell"));

# prints: A_1 A_11 A_12 A_2 B_1 B_11 B_12 B_2
# not what we want, in string sorting, "A_11" and "A_12" appears before "A_2";

# define a custom sorting function
my $my_well_sorting_func = sub {
  my @a = split /_/, $_[0];  # @a contains the wellRow and wellCol for the first well
  my @b = split /_/, $_[1];  # @b contains the wellRow and wellCol for the second well
  # sort wellRows first, then by wellCols
  my $res = ($a[0] cmp $b[0]) || (int($a[1]) <=> int($b[1]));
};

$t->sort("PlateWell", $my_well_sorting_func, 0);
print join(" ", $t->col("PlateWell"));

# prints the correct order: A_1 A_2 A_11 A_12 B_1 B_2 B_11 B_12
```

Method

`sort()`

7 Manipulating Two Tables

7.1 Joining two tables

Solution

```
$t = $t_product->join(      # $t_product is the first table
  $t_category,              # $t_category is the second table
  Data::Table::INNER_JOIN,  # table join type
  ["CategoryID"],           # primary key columns of the first table
  ["CategoryID"]            # primary key columns of the second table
);
```

Discussion

If the two tables are hosted by a database, one would use a SQL statement to join the two via foreign key-primary key connections. When the data are in files or in computer memory, it will be cumbersome to create database tables just in order to join the two tables. A operation similar to SQL JOIN can be performed using `join()` method. Similar to SQL, join supports INNER_JOIN (also known as JOIN or NATURAL_JOIN), LEFT_JOIN, RIGHT_JOIN, and FULL_JOIN. The key columns are defined as two column arrays, the reason for using an array is because a key in principle can comprise of multiple columns, such as a unique plate well is determined by the combination of its row ID and column ID. In INNER_JOIN, only those records, where keys are found in both tables are kept. In LEFT_JOIN, all records appear in the first table are kept, even if its key is missing in the second table (first parameter). RIGHT_JOIN is defined similarly to keep all records in the second table. FULL_JOIN will keep all records in either table, even if no match is found. Fields are marked as `undef` for those records missing their counter parts in the other table for non-INNER_JOIN mode.

Notice only the key columns from the first table are kept to avoid redundancy. All non-key columns from both tables are kept. If the two tables use the same column name for a non-key column, this will results in naming conflict in the resultant table, as we try to keep both columns. In that case, we would like to provide an additional fifth parameter, `{renameCol => 1}`, which will automatically rename conflicting column names in the second table by adding suffix _2 (or _3, _4, etc., if _2 has been taken).

Theoretically, NULL is not the same as NULL, so two NULL keys are not considered equal in the join, i.e., the fifth parameter is defaulted to `{matchNULL => 0}`. Practically, one might have a need to treat NULL's equal (especially when one of the multiple key columns is NULL), use `{matchNULL => 1}`. Be aware that NULL is not considered equal to an empty string. As Oracle treats NULL as empty string, so we also can set `{NULLasEmpty => 1}`. Obviously when NULLs are treated as empty strings, matchNULL is treated as 1.

Also be aware that the order of rows in the resultant table is not predictable. One might need an additional `sort()` to get resultant table into desirable order.

Method

`join(), Data::Table::INNER_JOIN, Data::Table::LEFT_JOIN, Data::Table::RIGHT_JOIN, Data::Table::FULL_JOIN`

7.2 Merging two tables row-wise

Solution

```
# $t_subset1 and $t_subset2 are two tables with the same set of columns
$t_subset1 = $t_product->match_pattern_hash('$_{ProductID}<=30');
print "Subset1: ".$t_subset1->nofRow."\n";
# Subset1: 30
$t_subset2 = $t_product->match_pattern_hash('$_{ProductID}>30');
print "Subset2: ".$t_subset2->nofRow."\n";
# Subset2: 47
# merge the second subset into the first subset, resulted back to the original table size
$t_subset1->rowMerge($t_subset2);
print "Subset1+Subset2: ".$t_subset1->nofRow."\n";
# Subset1+Subset2: 77
```

Discussion

When two datasets share the exact same columns and in the exact same column order, they can be merged row-wise, i.e., the rows from the second table are appended onto the first table vertically. The first table is modified. Be aware that if the columns of the second table are in a different order, the resultant table will contain incorrect data! To avoid this, give a second parameter `{byName => 1}`, this will merge the second table correctly by using its column names, rather than relying on its column positions. This parameter also solves another problem, when the second table misses a few columns. In that case, the missing element will be `undef`.

If the second table has extra columns when one uses {byName => 1}, the extra columns are ignored, because those columns are not part of the first table. However, if one intends to add those extra columns to the first table, use `{byName => 1, addNewCol =>1}`, all the elements of the new columns in the first table will be `undef`.

Method

`rowMerge()`

7.3 Merging two tables column-wise

Solution

```
# $t_subset1 and $t_subset2 are two tables with the same rows, but different columns
$t_subset1 = $t_product->subTable(undef, ["ProductID", "ProductName"]);
print "Subset1: ".join(",", $t_subset1->header)."\n";
# Subset1: ProductID,ProductName
$t_subset2 = $t_product->subTable(undef,["UnitPrice", "UnitsInStock"]);
print "Subset2: ".join(",", $t_subset2->header)."\n";
# Subset2: UnitPrice,UnitsInStock

$t_subset1->colMerge($t_subset2);
print "Subset1+Subset2: ".join(",", $t_subset1->header)."\n";
# Subset1+Subset2: ProductID,ProductName,UnitPrice,UnitsInStock
```

Discussion

When two datasets share the exact same records and in the same row order, they can be merged column-wise, i.e., the columns from the second table are appended onto the first table horizontally. The first table is modified. Be aware that if the rows of the second table are in different order, the resultant table will contain incorrect data. Typically, if record IDs are in both tables, one should use `join()`! `colMerge()` is used only when the two tables have exactly the same records in the same order.

If the second table contains column names already exist in the first table, we should use a second parameter `{renameCol => 1}` to fix the conflicts by renaming the duplicated column in the second table with a suffix _2 (or _3 if _2 already taken, etc.).

Method

`colMerge(), join()`

8 Transforming a Table

8.1 Reshaping – melting and casting for table statistics

Problem

A table contains observations for multiple objects and one often has to perform various statistics on it, a useful framework for such problems is called melting and casting.

Solution

```
# syntax
# melt(colsToGrp, colNamesToCollapseIntoVariable)
# cast(colsToGrp, colToSplit, colToSplitIsNumberOrString, colToFill, aggregationFunction)

# for two objects id = 1,2, we measure their x1 and x2 properties twice
$t = new Data::Table(
  [[1,1,5,6], [1,2,3,5], [2,1,6,1], [2,2,2,4]],
  ['id','time','x1','x2'],
  Data::Table::ROW_BASED);
# id    time    x1      x2
# 1     1       5       6
# 1     2       3       5
# 2     1       6       1
# 2     2       2       4

# first, melt a table into a tall-and-skinny table
# using the combination of id and time as the key
$t2 = $t->melt(['id','time']);
#id     time   variable value
# 1       1       x1      5
# 1       1       x2      6
# 1       2       x1      3
# 1       2       x2      5
# 2       1       x1      6
# 2       1       x2      1
# 2       2       x1      2
# 2       2       x2      4

# casting the table, &average is a method to calculate mean
$t3 = $t2->cast(['id'], 'variable', Data::Table::STRING, 'value', \&average);
# id      x1      x2
# 1       4       5.5
# 2       4       2.5
```

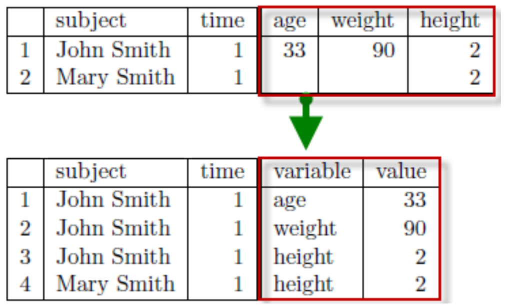
Discussion

Hadley Wickham introduced the melting-and-casting framework for common problems in data reshaping and aggregation. The framework is implemented in the "Reshape" package in R and a simplified version is available in `Data::Table`.

<u>Melting basically unpivots a table.</u> In this example, subjects (id = 1 and id = 2) were measured at time 1 and time 2 with two variables x1 and x2. Melting converts a short-and-wide table into a tall-and-skinny format, i.e., one specifies the columns for an identifier and the columns for measurement variables. In this case, a unique combination of id-time is the id, and x1, x2 are two variables. The variable array may be omitted if all non-id columns are treated as variables, i.e., the above `melt()` call is the same as

```
$t2 = $t->melt(['id','time'], ['x1', 'x2']);
```

As illustrated below (taken from the Reshape document[4]), the idea of melting is to convert the typical database table into a tall-and-skinny fact table. The purpose of melting is to enable different groupings, i.e., casting.



	subject	time	age	weight	height
1	John Smith	1	33	90	2
2	Mary Smith	1			2

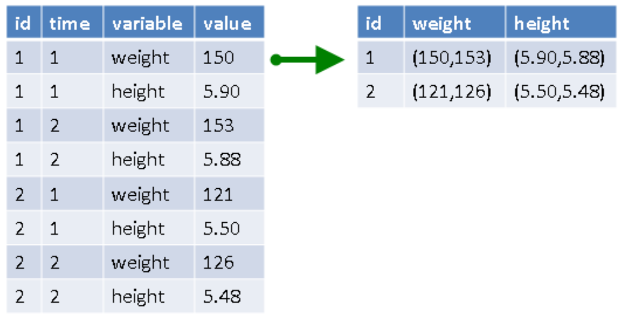
	subject	time	variable	value
1	John Smith	1	age	33
2	John Smith	1	weight	90
3	John Smith	1	height	2
4	Mary Smith	1	height	2

Casting is basically regrouping records into a contingency table. Here we choose `'id'` to be the row identifier and `'variable'` column contains data used to split `'value'` into different columns. As numerical values cannot be used as column names, this should be indicated in the third parameter, so that appropriate column names can be created. We expect to obtain a contingency table of id-by-x1,x2. There are probably multiple records share the same id-variable combination, therefore fall into the same destination cell, therefore these entries need to be aggregated using the supplied method. Currently the method can only return a scalar (the Reshape package in R can return multiple values) that will be placed in the resultant cell.

Let us repeat the casting process with another example below, where each id is measured twice for their weight and height. To regroup, we first define what will be our rows, i.e., unique id (group by id). Then we define the new column should be taken from the "variable", each unique value ("weight" and "height") becomes a new column. Then we fill the "values" into corresponding cells in the result table, i.e., each cell contains an array of "values" that match the row id and column header. Last we apply an aggregation method, say average, to each cell and generate the final result. The contribution of `melt()` is to restructure the data in such a way, that one can group data by id, by time, by id-time, etc. `Cast()` here is very similar to Excel's

[4] http://had.co.nz/reshape/introduction.pdf

pivot function. The GroupBy columns define the row, the Variable column defines columns, the Value column provides data for aggregation.



id	time	variable	value
1	1	weight	150
1	1	height	5.90
1	2	weight	153
1	2	height	5.88
2	1	weight	121
2	1	height	5.50
2	2	weight	126
2	2	height	5.48

id	weight	height
1	(150,153)	(5.90,5.88)
2	(121,126)	(5.50,5.48)

For the product table, if one would like to calculate total cost of products in each category, use

```
# modify UnitPrice column to store the UnitPrice*UnitsInStock value
map { $t_product->setElm($_,'UnitPrice',
        $t_product->elm($_,'UnitPrice')* $t_product->elm($_,'UnitsInStock'))
} 0 .. $t_product->nofRow - 1;
# notice the UnitPrice column now actually stores total cost
print $t_product->cast(['CategoryID'], undef, undef, 'UnitPrice', \&sum)->csv(1);

CategoryID,(all)
1,12480.25
2,12023.55
7,3549.35
...
```

The first parameter indicates we would like to group records by `CategoryID`, since we do not have a variable column that contains data to be made into columns, we provide `undef` for the next two parameters. In this case, `cast()` will create only one column called `'(all)'` (using the Reshape package convention) and throw all recordings sharing the same `CategoryID` into that destination cell and call the aggregation function `sum`. Notice, we pass UnitPrice column to sum subroutine, where UnitPrice already contains the total cost of each product.

Let us look at another example, where we start with an employee salary table and try to calculate average salary for different groupings.

```
$t = new Data::Table( # create an employ salary table
    [
      ['Tom', 'male', 'IT', 65000],
      ['John', 'male', 'IT', 75000],
      ['Tom', 'male', 'IT', 65000],
      ['John', 'male', 'IT', 75000],
      ['Peter', 'male', 'HR', 85000],
      ['Mary', 'female', 'HR', 80000],
      ['Nancy', 'female', 'IT', 55000],
      ['Jack', 'male', 'IT', 88000],
      ['Susan', 'female', 'HR', 92000]
    ],
    ['Name', 'Sex', 'Department', 'Salary'], Data::Table::ROW_BASED);

# get a Department x Sex contingency table, get average salary across all four groups
# Department defines the row, Sex defines the column, Salary fills the cells for average
print $t->cast(['Department'], 'Sex', Data::Table::STRING, 'Salary', \&average)->csv(1);
Department,female,male
IT,55000,73600
HR,86000,85000
# get average salary for each department
# Department defines the row, '(all)' is the column, Salary fills the cells for average
print $t->cast(['Department'], undef, Data::Table::STRING, 'Salary', \&average)->csv(1);
Department,(all)
IT,70500
HR,85666.6666666667

# get average salary for each gender
# Sex defines the row, '(all)' is the column, Salary fills the cells for average
print $t->cast(['Sex'], undef, Data::Table::STRING, 'Salary', \&average)->csv(1);
Sex,(all)
male,75500
female,75666.6666666667

# get average salary for all records
# All records go into one row, '(all)' is the column, Salary fills the cells for average
print $t->cast(undef, undef, Data::Table::STRING, 'Salary', \&average)->csv(1);
(all)
75555.5555555556
```

Method

`melt(), cast()`

8.2 Grouping a table with aggregation functions

Solution

```
# syntax
# group(colsToGrp, colsToAggregate, AggregateFunctions, resultColNames)
$t = $t_product->group(
       ['CategoryID'],
       ['UnitPrice', 'UnitsInStock'],
       [\&average, \&average],
       ['Avg(UnitPrice)','Avg(UnitsInStock)'],
       0);
print $t->csv(1);

#output
CategoryID,Avg(UnitPrice),Avg(UnitsInStock)
1,37.9791666666667,46.5833333333333
2,23.0625,42.25
7,32.37,20
...
```

Discussion

Group all rows based on their primary key columns (first parameter), for each group, we take a few columns (second parameter) and apply individual aggregate methods (third parameter). The resultant values are the new table elements. The new column names are specified in the fourth parameter. To silent the output of all remaining columns, use 0 as the fifth parameter (`$keepRestCols = 0`).

If one set $keepRestCols to 1, the value of the first encountered row for that primary key is used. Using the same employee salary table as introduced in the Discussion section of 8.1, we can do the same statistics using group() instead of cast().

```
# get average salary for each department
print $t->group(['Department'], ['Salary'], [\&average], ['Avg(Salary)'])->csv(1);
Department,Avg(Salary)
IT,70500
HR,85666.6666666667

# get average salary for all employees
print $t->group(undef, ['Salary'], [\&average], ['Avg(Salary)'])->csv(1);
Avg(Salary)
75555.5555555556

# get average salary for each department-sex combination
print $t->group(['Department', 'Sex'], ['Salary'], [\&average], ['Avg(Salary)'])->csv(1);
Sex,Department,Avg(Salary)
male,IT,73600
male,HR,85000
female,HR,86000
female,IT,55000
```

In the last example, where we group by both Department and Sex, the resultant table is in a tall-and-skinny format, not in the contingency format as what is produced by cast(). Actually cast() does the combination of group() and pivot(); pivot() is described next. In general, group() might be easier to undertand than cast().

Method

group(), pivot(), cast(), melt()

8.3 Pivoting a table

Problem

Convert a tall-any-skinny format table into a fat table.

Solution

```
# Syntax
# pivot(colToSplit, colToSlitIsStringOrNumber, colToFill, colsToGrp)
$t = new Data::Table(
  [[1,1,1,1,2,2,2,2],
   [1,1,2,2,1,1,2,2],
   ['x1','x2','x1','x2','x1','x2','x1','x2'],
   [5,6,3,5,6,1,2,4]],
  ['id','time','variable','value'],
  Data::Table::COL_BASED);
# $t is a tall-and-skinny table
#id     time   variable value
# 1       1       x1      5
# 1       1       x2      6
# 1       2       x1      3
# 1       2       x2      5
# 2       1       x1      6
# 2       1       x2      1
# 2       2       x1      2
# 2       2       x2      4

# id-time defines the row, variable defines the columns, value used to fill the cells
$t2 = $t->pivot('variable', Data::Table::STRING, 'value', ['id','time']);
print $t2->csv(1);
# output
# id,time,x1,x2
# 1,1,5,6
# 1,2,3,5
# 2,1,6,1
# 2,2,2,4
```

Discussion

The word "pivot" means a similar thing as the pivot function in Excel, except it does not aggregate. Given a tall-and-skinny table, `pivot()` takes all the unique values in a given column (defined in the first parameter, `colToSplit`, `variable` in this example) and make each of those values an independent column in the resultant table. Undefined value is treated the same as string "NULL". If the column "XYZ" consists of numeric values (column type for the `$colToSplit` is defined by the second parameter, either `Data::Table::NUMBER` or `Data::Table::STRING`), the column name is "XYZ=<value>", as numbers cannot be used as column names.

The `pivot()` method first groups all rows based on their primary key columns (defined as the fourth parameter), and records are further put into cells, where their values match the `colToSplit` value of that column (say <value>). That is each row fall into the cell defined by the <key>-<value> coordinate, where row position is determined by <key> and column position is determined by <value>. For all the records fall into a cell, their `colToFill` value (defined in the third parameter) is used to fill the cell. Theoretically, there could be multiple rows share the same <key>-<value> pair, therefore end up into the same cell. In case a collision happens, only one record is kept. In real life datasets, such collision rarely happens, so we do not implement more complicated methods for dealing with such collisions. If we encounter a collision, it means we should have applied `group()` to our data, before `pivot()`, but if that is the case, we should really use `cast()` instead. If `colToFill` is not specified, the cell will contain the number of records fall into that cell. If primary key columns are not specified, all rows are considered to share the same key, i.e., only one row is output.

The method can take a fifth parameter, `$keepRestCols`. If set, all remaining columns are also exported (if multiple values for a given key are found, only one value is output). However, this is rarely used.

The idea of pivot is probably explained better in the cast method, see 8.1.

In general, `cast()` is the combination of `group()` and `pivot()`, therefore is equivalent to the pivot in Excel. The `pivot()` in this package assumes `group()` has been applied before, therefore it does not deal with aggregation itself. If a table is not ready for `cast()`, `melt()` (i.e., unpivot) should be applied first.

Method

`pivot(), group(), cast(), melt()`

8.4 Looping through groups

Solution

```
$t_product->each_group(
  ['CategoryID'],
  sub {
    my $t = shift; # the first parameter is a table of all records of the same key
    print "Category ID: ".$t->elm(0,'CategoryID').", has ".$t->nofRow." products.\n";
  }
);

Outputs:
Category ID: 1, has 12 products.
Category ID: 2, has 12 products.
Category ID: 3, has 13 products.
...
```

Discussion

Sometimes, it is quite useful to break the records into groups and apply a method for each group. This can be achieved by `each_group()` method, which takes an array reference that defines the primary key columns, followed by a user-defined calculation method that will be applied.

The user-defined calculation method can take two parameters, the first parameter is a `Data::Table` object that consists of all the rows share a certain key (key can be obtained from the first row of this table, so we do not pass in keys). The optional second parameter can obtain an array reference, which points to a list of row indices for all the matched rows in the original table, in case one needs to use that to access and modify the original table.

The example above simply groups all products by its `CategoryID` and count the size of each group. At this point, the keys are sorted first and then the calculation method is applied in that order.

Method

`each_group()`

9 Shortcuts

9.1 Useful constants

Discussion

When call `sort()`, one has to specify whether a column is a string type or a number type, and whether it should be sorted in ascending or descending order. The parameters are specified as 1 or 0, where 0 represents numeric and ascending, 1 represents string and descending. It is hard to remember, therefore, we are encouraged to use constants `Data::Table::NUMBER`, `Data::Table::STRING`, `Data::Table::ASC`, `Data::Table::DESC` instead.

For the same reason, `Data::Table::ROW_BASED` and `Data::Table::COL_BASED` are useful in `new()` when initializing an table.

For `join()` method, we should use constants: `Data::Table::INNER_JOIN`, `LEFT_JOIN`, `RIGHT_JOIN`, or `FULL_JOIN`.

When we use `fromCSV()` or `fromTSV()`, sometimes we need to specify the operating systems that the file was generated in. Constants `Data::Table::OS_UNIX`, `OS_PC`, and `OS_MAC` could be handy.

`NUMBER` and `STRING` are useful for any method that expects such a column type parameter, such as in `pivot()`.

Method

```
Data::Table::NUMBER, Data::Table::STRING, Data::Table::ASC,
Data::Table::DESC, Data::Table::ROW_BASED, Data::Table::COL_BASED,
Data::Table::INNER_JOIN, Data::Table::LEFT_JOIN,
Data::Table::RIGHT_JOIN, Data::Table::FULL_JOIN, Data::Table::OS_UNIX,
Data::Table::OS_PC, Data::Table::OS_MAC
```

9.2 Sugar

Discussion

`isEmpty()` is a shortcut for `$t->nofRow == 0`.

`hasCol($colID)` is a shortcut for `$t->colIndex($colID) >= 0`.

`lastRow()` is a shortcut for `$t->nofRow - 1`.

`lastCol()` is a shortcut for `$t->nofCol - 1`.

`colName($colNumericIndex)` is a shortcut for `($t->header)[$colNumericIndex]`.

Method

`isEmpty(), hasCol(), lastRow(), lastCol(), colName()`

10 Pitfalls

10.1 Internal data structures

It is unclear what the best data structure for `Data::Table` should be. In C#, DataTable is implemented with a row-based data structure, where each `DataRow` is a record of multiple fields. A row conceptually is an object, a class instance, therefore, such a data structure matches database concept well and is quite efficient for object-based operations, i.e., best for frequently add and delete records. To find the `UnitPrice` of a product, one first fetch the product row, then look up its `UnitPrice` field.

For scientific computing, one often has to operate on columns, say calculating the average `UnitPrice`, or to add a new column `TotalCost` based on `UnitPrice` and `UnitsInStock`. As data in a column are of the same type, which can be stored as a native array and this structure are most efficient for numerical processing, as well as most efficient for frequent adding and deleting of columns, such as the data.frame data type in R.

As mentioned above, it will be the most efficient to organize data by columns, if add or delete columns are expected to be frequent. It will be the most efficient that the data are row-based, if add or delete objects are expected to be frequent. So the performance really depends on the use cases. After quite some researches and debates, original authors of this module decided to implement `Data::Table` with both data structures.

Data internally may be stored as row-based at one moment, but column-based at another. There is an internal method called rotate() does such transformation. Sometimes it is for efficiency, i.e., after one `delCol()` method call, the implementation is switched into col-based to anticipate another column operation, so that the next deletion will be faster. If one `addRow()`, the implementation becomes a row-based, so that the next `addRow()` operation would be faster. Sometimes, a method is implemented with a rotate call first, not for the sake of performance consideration, but for the cleanness of the coding. If there is no obvious advantage of transforming the data structure into one or another form, a method might be implemented to support both structures (such as `elm()`), so that expensive rotate() operation is avoided.

So the best is to do multiple row-based operations and col-based operations together in a group. The `Data::Table` may waste time in rotating back and forth, if you frequently interlace row and column operations. Most users should not make any assumption on the internal data representation. Only for those who are really familiar with the rotate concept may try to take advantage of internal representation, when it comes to process really large datasets.

10.2 Volatile methods

Typically operations such as `col()` or `row()` returns a copy of table elements, as users are likely going to modify that data. The copy can be expensive, if we handle a large-scale table. To avoid copying, we provide some methods `rowRef()`, `rowRefs()`, `colRef()`, `colRefs()`, `elmRef()`, and `data()` (reference to the data matrix used inside the table). These methods return the reference pointing into the existing table.

When `colRef()` method returns, it makes sure the table data are organized in a col-based matrix and returns a reference to the column array. Because, the data structure in the table can be rotated between row-based and col-based as discussed in 10.1, a column reference will become staled, if you call a row-based operation that triggers a rotation, before you use the previously returned column reference. Therefore one has to be extremely cautious. Typically, one should consume the reference, before the next `Data::Table` call is issued. If you are not familiar with what method call results in row/column-based data structure, avoid using these reference methods altogether. We can think of these methods are for the author's own use in various other internal method implementations. Note: `$t->type()` tells us whether the table object is row/col-based at the moment.

Method

```
rowRef(), rowRefs(), colRef(), colRefs(), elmRef(), data(), type()
```

Index

%Data::Table::DEFAULTS 9
addCol() 22
addRow() 21, 41
cast() 36, 37, 39
col() 19
colIndex() 17, 26
colMerge() 32
colName() 41
colRef() 42
colRefs() 42
CSV() 12
data() 42
Data::Table::ASC 40
Data::Table::COL_BASED 25, 40
Data::Table::DESC 40
Data::Table::FULL_JOIN 31, 40
Data::Table::INNER_JOIN 31, 40
Data::Table::LEFT_JOIN 31, 40
Data::Table::NUMBER 40
Data::Table::OS_MAC 40
Data::Table::OS_PC 40
Data::Table::OS_UNIX 40
Data::Table::RIGHT_JOIN 31, 40
Data::Table::ROW_BASED 25, 40
Data::Table::STRING 40
delCol() 22, 41
delCols() 22
delRow() 21
delRows() 21
each_group() 39
elm() 17
elmRef() 42
fromCSV() 9, 12
fromFile() 10
fromSQL() 11
fromTSV() 9
group() 37, 39
hasCol() 17, 41
html() 15
html2() 15
isEmpty() 18, 41
iterator() 19
join() 31, 32
lastCol() 18, 41
lastRow() 18, 41
match_pattern() 26, 27, 28, 29
match_pattern_hash() 26, 27, 28, 29
match_string() 26, 27, 28, 29
melt() 36, 37, 39
moveCol() 23
new 24, 25
nofCol() 18
nofRow() 18
pivot() 37, 39
rename() 24
reorder() 23
replace() 23
rotate() 41
row() 19
rowHashRef() 19
rowMask() 28
rowMerge() 32
rowRef() 42
rowRefs() 42
setElm() 19, 20, 27
sort() 30
subTable() 29
tables2xls() 16
tables2xlsx() 16
tsv() 14
type() 42
unpivot 33
xls2tables() 13
xlsx2tables() 13