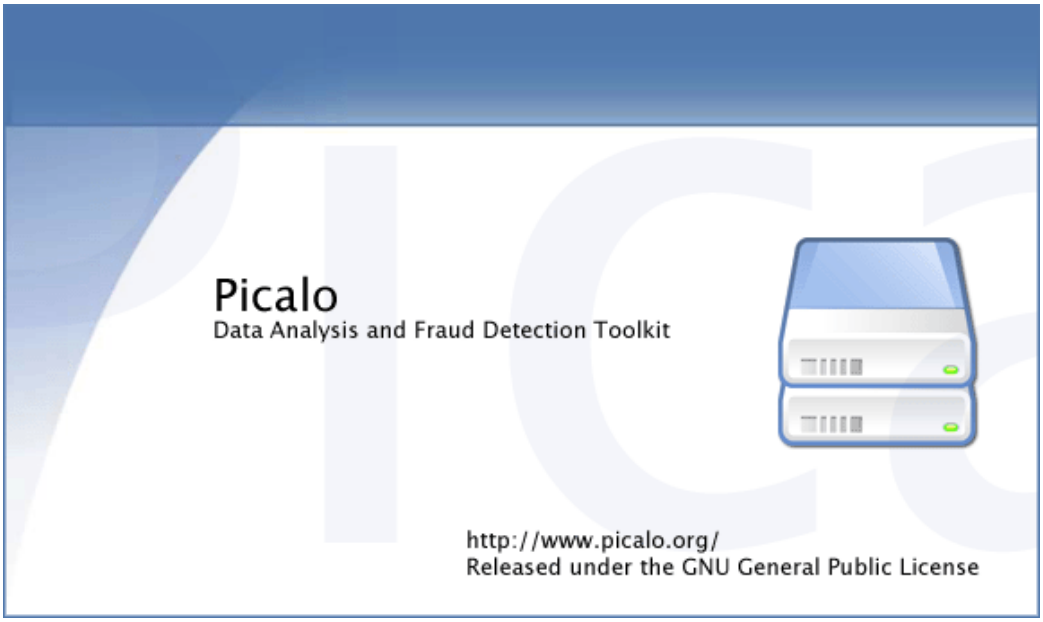

Picalo Cookbook

Conan C. Albrecht, PhD

March 31, 2010

Contents

About This Book

The purpose of this book is to show you how to accomplish simple and advanced tasks in Picalo. It is structured in the familiar "cookbook" format, with recipes for different things you might need to do.

Each recipe in the book shows both the GUI and scripting ways to accomplish each task. While most tasks can be performed either way, it is noted when tasks are only available with scripting commands. To speed the writing of this book, screen shots have been omitted in favor of textual descriptions.

Scripting commands can be entered two ways into Picalo. First, you can simply type them line by line into the Shell at the bottom right of your screen. Second, you can create a new script, type all the command lines, and click the Run button on the toolbar.

Contributions

I welcome contributions to this manual. If you have a recipe to submit, email the text to me and I'll attribute it to your name in the text.

A Moving Target

Since Picalo is a work in progress, it is constantly improving and changing. Therefore, the dialogs in the program may be slightly different that what you see in this manual due to change in the program. We'll try to keep the manual updated with the program, but we simply do not have enough time to update the manual in perfect unison with the program. Since the Picalo program itself is the main priority, the manual may be slightly behind the actual features or user interface of the program.

Chapter 1

Working With Tables

This chapter presents recipes for working with tables, from creating new tables to adding records and columns.

Tables are the basic data type of Picalo, and they are the most important thing for you to understand. All data are kept in Picalo table objects. Almost all Picalo functions input and output one or more tables.

Tables are made up of records and columns. Records are sequentially numbered. Columns are named and have explicit types. Columns can be displayed with format masks, and they can also be calculated.

Tables always have a name in Picalo, which may be different from the actual filename the table was loaded from. Names are important so the table can be referenced in the Shell, in scripts, and in dialog boxes. The script examples in this section generally use *data* for the table name.

1.1 Create a Table

Picalo GUI Recipe:

1. Select *File — New Table...*
2. Enter a name for the new table (must follow the rules for column names).
3. Enter the column names and types into the dialog.
4. Click the *Save* button.

Script Recipe:

```
data = Table([
  ( 'ID', unicode ),
  ( 'Name', unicode ),
  ( 'Age', int),
  ( 'BirthDate', Date ),
  ( 'Salary', float ),
])
```

Discussion:

Creating a new table requires defining its column names and types. Column names must start with a regular letter (A-Z, a-z, or underscore) and then contain any combination of letters or numbers (A-Z, a-z, underscore, 0-9). Column names *are* case sensitive, meaning the columns myID and MYid and MyId are different names. The following names are valid:

- ID
- myID
- FirstName
- First_Name

- _first_name
- col8

The following names are invalid and will be rejected by Picalo:

- 8col (starts with a number)
- First Name (contains a space)
- int (Picalo reserved word)
- Age*Name (contains a star)

Picalo supports the following column types:

- str - a string column of any length
- unicode - a string that supports international characters, up to 65,535 characters in length
- int - an integer in the range -2,147,483,647 to 2,147,483,647 (actual range depends upon platform)
- long - a long integer with greater range than int (actual range depends upon platform)
- number - a floating-point number (i.e. has a decimal)
- Date - a date field with various formats
- DateTime - a date and time field with various formats

Picalo can actually hold more types than those listed above, but these are the types supported directly by the GUI. You can use any Python type if you create tables from the Shell. If you are wondering, there is a reason the Date and DateTime types are capitalized and the others are not.

1.2 Access a Cell Value

Picalo GUI Recipe:

1. Double-click a table in the left-side project browser to view table data.

Script Recipe:

```
print data[0].Age                 # value in the first record, Age column
print data[0]['Age']              # value in first record, Age column
print data['Name'][4]             # value in the fifth record, Name column
print data.column('Name')[4]      # value in the fifth record, Name column
print data[0][2]                  # value in the first record, third column
print data.record(0).column(2)    # value in the first record, third column
print data[-1].Name               # value in the last record, Name column
```

Discussion:

Picalo supports a wide variety of methods for accessing table cell values to support different coding styles. The first script line is the preferred Picalo code for accessing records.

As with most things in Picalo, records indices are zero-based, which means that the first record is [0], the second record is [1], and so forth. Record indices also support negative numbers, which number from the bottom of the table. Picalo is zero-based because it's base language, Python, is zero-based. New users who have not used zero-based languages like Java, C++, or C# before may feel uncomfortable with this at first.

The script recipe section above shows a number of ways to assign the cell value to a variable. See the comments on each line for a description of which cell is being accessed.

1.3 Modify a Cell Value

Picalo GUI Recipe:

1. Double-click a table in the left-side project browser to view the table.
2. Start typing in the cell you wish to modify.

Script Recipe:

```
1 data[0].Age = 13
2 data[0] = [ 'Fred', 'Smith', 13 ]
```

Discussion:

Use the equals sign to assign a value to a cell. The format of the left side of the equation follows the same formats available when accessing a cell value. See the previous section for various formats.

The second script line shows how to modify an entire record at once by setting the record equal to a list (denoted with square brackets and commas).

Whenever a cell value is set, the new value is coerced into the right type using the column type. For example, if you set the value of a cell to the string '13' and the column type is *int*, Picalo will convert the value to the integer 13. If Picalo cannot coerce the value into the right type, you will get an error object in the field.

1.4 Set a Cell to None

Picalo GUI Recipe:

1. Not available (yet).

Script Recipe:

```
1 data[0].Age = None
```

Discussion:

The *None* type in Picalo represents an empty value. This is different than a zero value (which means a value of zero :) or an empty string (”). The None value means that nothing exists in a cell. It is similar to the null type in other programs.

Picalo defines the keyword *None* (without quotes) to represent this non-existent value. None values can throw off analyses if you are not careful.

1.5 Retrieve a Table Record

Picalo GUI Recipe:

1. Double-click a table in the left-side project browser to view table data.
2. Browse to the record you want to retrieve.

Script Recipe:

```
1 rec = data[0]
2 rec[0] = 'Fred'
3 rec.Age = 13
4 print rec.Name
5
6 for cell in rec:
7     print cell
```

Discussion:

Picalo records are objects that look like lists of data. As such, you can set entire records equal to a variable name and access the columns of the given record using the familiar idioms (by index or by name).

As a list-like object, you can iterate over a record and access the individual cells using a *for* loop, as the example shows.

1.6 Retrieve Several Table Records By Index

Picalo GUI Recipe:

1. Open the source table into the viewer window.
2. Select *Data — Select — By Record Index* .
3. Enter the starting and ending row indices, enter a results table name, and press OK. The starting and ending indices are inclusive in this dialog.

Script Recipe:

```
# retrieve records 5-9 inclusive
newtable = data[5:10]

# retrieve every other record 5-9 inclusive
newtable2 = data[5:10:2]

# retrieve the first 10 records of the table
newtable3 = data[:10]

# retrieve all records after record 14
newtable4 = data[15:]

# retrieve all records except the last 20
newtable5 = data[0:-20]
```

Discussion:

You can retrieve several records from a table using *slices*. A slice has a starting index and ending index, separated by a colon. The result is a new table containing the given records.

Consistent with the Python language, the slice is inclusive of the first number and exclusive of the second number. In other words, the first example above (newtable) retrieves records 5-9 inclusive, or the sixth, seventh, eighth, and ninth rows.

A third number can be added to specify a stepping value for the indices. In the second example above (newtable2), records 5, 7, and 9 (sixth, eighth, and tenth table rows) are copied into the new table.

Picalo allows you to leave off the starting or ending index, as shown in the third and fourth example. If you leave off the first index (third example), Picalo will assume the beginning of the table (index 0). If you leave off the last index (fourth example), Picalo will assume the end of the table (last record). If you leave off both indices [:], Picalo will copy the entire table to another table.

Finally, you can use a negative index to count from the end of the table. In the fifth example, the -20 means the last 20 records of the table. Assuming the data table has 100 records, [0:-20] is identical to [0:80].

1.7 Retrieve a Table Column

Picalo GUI Recipe:

1. Column objects are not directly available in the GUI, but their names and types can be modified in the Table Properties dialog.

Script Recipe:

```
1 col = data['Name']  # retrieve the name column
2 col[0] = 'Danny'
3 print col.get_type()      # prints <str>
4 print col.get_name()    # prints 'Name'
5
6 for cell in col:
7     print cell
```

Discussion:

Similar to record objects, column objects act like regular lists of values. You can set individual record cells for the column using square brackets and the record index. Columns also have a number of object methods to retrieve column type, name, and so forth.

As a list-like object, you can iterate over a column and access the individual cells using a *for* loop, as the example shows.

1.8 Add a Record to a Table

Picalo GUI Recipe:

1. Double-click a table in the left-side project browser to view table data.
2. Navigate to a record below where you wish to add the new record.
3. Select *File — Row — Insert Row* (or *Append Row* for the end of the table).
4. The cells in the new row will be set to None.

Script Recipe:

```
1  data.insert(2, 'Fred', 'Smith', 35)  # insert before third record with cell values
2
3  data.insert(2)                       # insert before the third record with None values
4
5  rec = data.insert(2)                 # insert before the third record with None values
6  rec.FirstName = 'Fred'               # set the values of the record
7  rec.LastName = 'Smith'
8  rec.Age = 35
9
10 data.append('Fred', 'Smith', 13)     # append to table end with cell values
11
12 data.append()                        # append to table end with None values
13
14 rec = data.append()                  # append to table end with None values
15 rec.FirstName = 'Fred'               # set the values of the record
16 rec.LastName = 'Smith'
17 rec.Age = 35
```

Discussion:

New records can be inserted or appended anywhere in a table. The script code shows several possible ways of doing this, depending upon your coding style.

1.9 Delete a Record from a Table

Picalo GUI Recipe:

1. Double-click a table in the left-side project browser to view table data.
2. Navigate to the record you wish to delete.
3. Select *File — Row — Delete Row...*

Script Recipe:

```
1 del data[5]      # deletes the sixth record in the table
2 del data[-1]     # deletes the last record in the table
```

Discussion:

Picalo follows the standard Python way of deleting list items using the *del* keyword. As with all table operations, you can use a negative index to start at the end of the table.

1.10 Count the Records in a Table

Picalo GUI Recipe:

1. Double-click a table in the left-side project browser to view table data.
2. The number of records in the table will show in the bottom-right of the Picalo window.

Script Recipe:

```
1 numrecs = len(data)
```

Discussion:

Following the standard Python syntax, use the *len* function to count the number of items in any list, including in a Picalo table. Note that if you retrieved a large data set from a database, calling this method may pull all the records into memory so Picalo can count the number of records. Some databases can return the length of a result set directly, while others require Picalo to count manually.

1.11 View Table Column Names

Picalo GUI Recipe:

1. Double-click a table in the left-side project browser to view table data.
2. Select *File — Table Properties...* or click the 'wrench' icon on the toolbar.
3. The table properties dialog shows, letting you view the columns in the table.

Script Recipe:

```
# print the column names
print data.get_column_names()

# iterate across the rows of a table, then the columns of each row
for rec in data:
    for colname in data.get_column_names():
        print rec[colname]

# retrieve the actual column objects
cols = data.get_columns()
for col in cols:
    print col.get_name()
```

Discussion:

There are a number of ways to access the column names in a table. The primary way is the *get_column_names()* method of table, which returns a list of column names.

An alternative way is to access the actual column objects, as shown in the last example. This gives you access to the methods of the column, including *get_name()*, *get_type()*, and *get_format()*.

1.12 View Table Structure

Picalo GUI Recipe:

1. Double-click a table in the left-side project browser to view table data.
2. Select *File — Table Properties...* or click the 'wrench' icon on the toolbar.
3. The table properties dialog shows, letting you view the structure of the table.

Script Recipe:

```
1 struct = data.structure()
2 struct.view()
```

Discussion:

The *structure()* method of table returns a new table containing the column information for the entire table, including the column names, types, expressions, and format.

1.13 Change a Column Name

Picalo GUI Recipe:

1. Double-click a table in the left-side project browser to view table data.
2. Select *File — Table Properties...* or click the 'wrench' icon on the toolbar.
3. The table properties dialog shows. Change the column name and click 'Save'.

Script Recipe:

```
1 # first get the column object, then change its name
2 data['Name'].set_name('FirstName')
3
4 # change the name using the table method
5 data.set_name('Name', 'FirstName')
```

Discussion:

Column names can be changed at any time, using the GUI, the column object, or the table itself. Column names must be unique and must follow the rules described earlier in this chapter.

1.14 Change a Column Type

Picalo GUI Recipe:

1. Double-click a table in the left-side project browser to view table data.
2. Select *File — Table Properties...* or click the 'wrench' icon on the toolbar.
3. The table properties dialog shows. Change the column type and click 'Save'.

Script Recipe:

```
1 # first get the column object, then change its type
2 data.column('Name').set_type(unicode)
3
4 # change a type using the table method
5 data.set_type('Salary', float)
```

Discussion:

Column types can be changed at any time, using the GUI, the column object, or the table itself. The existing values in the column will be immediately coerced to the new type. If values cannot be converted, an error object will be placed in the cell.

Picalo supports the following column types:

- str - a string column of any length
- unicode - a string that supports international characters, up to 65,535 characters in length
- int - an integer in the range -2,147,483,647 to 2,147,483,647 (actual range depends upon platform)
- long - a long integer with greater range than int (actual range depends upon platform)

- float - a floating-point number (i.e. has a decimal)
- Date - a date field with various formats
- DateTime - a date and time field with various formats

Picalo can actually hold more types than those listed above, but these are the types supported directly by the GUI. You can use any Python type if you create tables from the Shell. If you are wondering, there is a reason the Date and DateTime types are capitalized and the others are not.

1.15 Change a Column Format

Picalo GUI Recipe:

1. Double-click a table in the left-side project browser to view table data.
2. Select *File — Table Properties...* or click the 'wrench' icon on the toolbar.
3. The table properties dialog shows. Click on one of the column names to display its information on the right. Picalo can format date and number fields.
4. In the format text box, enter a format appropriate to the column type (such as `$#,##0.00` for currency).
5. Alternatively, click the little 'gear' icon next to the format text box to open the format dialog. This dialog will help you create the format.
6. Click the 'Save' button to finish.

Script Recipe:

```
1 # set the amount column to have three decimal places
2 data['Amount'].set_format('0.000')
3
4 # set the format using the table method of a date column
5 data.set_format('Birthdate', '%Y-%m-%d')
```

Discussion:

Formats in Picalo are very important for number and date fields. They affect how values are entered into a field and how values are displayed. The format dialog (accessed from the table properties dialog) is useful when putting formats together.

Date Fields

Picalo supports two types of date fields: Date and DateTime. It uses the standard format characters to both parse input values and format display

values. In other words, Picalo will use the format to interpret data you enter into a table, whether by code, the GUI, or import. Be sure to set the format correctly before you start entering data.

The date specifiers are as follows:

- %b Locale's abbreviated month name.
- %B Locale's full month name.
- %c Locale's appropriate date and time representation.
- %d Day of the month as a decimal number [01,31].
- %H Hour (24-hour clock) as a decimal number [00,23].
- %I Hour (12-hour clock) as a decimal number [01,12].
- %j Day of the year as a decimal number [001,366].
- %m Month as a decimal number [01,12].
- %M Minute as a decimal number [00,59].
- %p Locale's equivalent of either AM or PM. (1)
- %S Second as a decimal number [00,61]. (2)
- %U Week number of the year (Sunday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Sunday are considered to be in week 0. (3)
- %w Weekday as a decimal number [0(Sunday),6].
- %W Week number of the year (Monday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Monday are considered to be in week 0. (3)
- %x Locale's appropriate date representation.
- %X Locale's appropriate time representation.
- %y Year without century as a decimal number [00,99].
- %Y Year with century as a decimal number.

- %Z Time zone name (no characters if no time zone exists).
- %% A literal '%' character.

Following are examples of date formats:

- 2009-01-31 — %Y-%m-%d
- January 25, 2015 — %B %d, %Y
- 10:Jan:31 — %y-%b-%d
- January 25, 2015, 15:30:22 — %B %d, %Y, %H:%M:%S

Number Fields

Picalo can format numbers as currency, in scientific notation, etc. Number formats are different from date formats because they do not affect the input of data; they only affect the display. For numbers you enter via the GUI or import from text files, Picalo uses a pretty intelligent parser to pull the number out of the field. It automatically discards monetary signs, commas, and other extra characters. It can automatically detect scientific notation.

Number formats are important for display on the screen, printed reports, and exported files. The special characters are as follows:

- Prefix the format with any character(s) (like $) to add to the front of the number.
- End the format with a zero (0) to round to the nearest whole number.
- Use a period (.) to denote decimal portions of the number.
- Use a pound (#) to denote a regular number.
- Use a comma (,) to denote a thousands separator (use with # signs to place it every three numbers)
- Use a percent (%) to denote the value should be displayed as a percent (multiplied by 100 to when displayed and divide by 100 when parsing input and the number ends with a %)
- Use the letter E+ followed by zeros to denote scientific notation.

Following are examples of number formats:

- 0 - Rounds to the nearest whole number. 10.99 is displayed as 11; 12.3 is displayed as 12.
- 0.00 - Rounds to the nearest hundredth. 10.99 is formatted as 10.99; 12.309 is formatted as 12.31; 13 is formatted as 13.00.
- $0.00 - Rounds to the nearest hundredth and adds a dollar sign to the front of the number. (you can also use any other character, such as the euro glyph)
- #,### - Formats thousands with a comma. 1234.56 is formatted as 1,234.56.
- #,##0 - Formats thousands with a comma and rounds to the nearest whole number. 1234.56 is formatted as 1,235.
- #,##0.000 - Formats thousands with a comma and rounds to the nearest thousandth. 1234.56 is formatted as 1,234.560.
- 0% - Rounds to the nearest whole number, multiplies by 100 (for display only), and adds a percent sign.
- 0.00% - Rounds to the nearest hundredth, multiplies by 100 (for display only), and adds a percent sign.
- #E+000 - Shows the number in scientific notation to the given number of decimal places.

1.16 Add a Column to a Table

Picalo GUI Recipe:

1. Double-click a table in the left-side project browser to view table data.
2. Select *File — Table Properties...* or click the 'wrench' icon on the toolbar.
3. The table properties dialog shows. Navigate above or below the column you want to add the new column by.
4. Click the 'star' icon on the left side of the dialog. One icon will add the new column above the current column; the other will add below.
5. Set the new column name and type. Click 'Save' to finish.

Script Recipe:

```
# add a column to the end of a table
data.append_column('FavColor', unicode)

# add a column to the end of a table, with initial values for each row
data.append_column('FavColor', unicode, [
   'Yellow',
   'Yellow',
   'Blue',
   'Green',
  # additional values for each record
])

# insert a column in the third position
data.insert_column(2, 'FavColor', unicode)
```

Discussion:

New columns initialize with the None type in each cell, unless you provide initial values for the column. When a column is inserted, as in the last example, all other columns are pushed right to make room for it.

1.17 Add an Active Calculated Column to a Table

Picalo GUI Recipe:

1. Double-click a table in the left-side project browser to view table data.
2. Select *File — Table Properties...* or click the 'wrench' icon on the toolbar.
3. The table properties dialog shows. Navigate above or below the column you want to add the new column by.
4. Click the 'star' icon on the left side of the dialog. One icon will add the new column above the current column; the other will add below.
5. Set the new column name. Set the column type to one of the types.
6. Add a calculation in the form of an expression. If you click the expression builder, a dialog will help you build the expression. Following are example expressions:
   - *Double the amount column*: `Amount * 2`
   - *Add two columns*: `UnitPrice + ItemTax`
   - *Use a Picalo function*: `Benfords.get_expected(Amount, 2)`
7. Check the "Active Calculation" box.
8. Click the 'Save' button to finish.

Script Recipe:

```
# add a calculated column to the end of a table
data.append_calculated('Amount2', int, 'Amount * 2')

# add two columns; insert new col before Name column
data.insert_calculated('Name', 'Subtotal', number, 'UnitPrice + ItemTax')

# use a Picalo function; insert new col at the beginning of table
data.insert_calculated(0, 'Benford', number, 'Benfords.get_expected(Amount, 2)')

# insert a random number
import random
data.append_calculated('RandomNum', number, 'random.randint(1, 10)')
```

Discussion:

The above methods insert *active* calculated columns, meaning that the results will constantly update and change as the source data change. If you change the Amount cell, the first calculated column above (Amount2) will immediately update to reflect the new value.

The final example above (random number) will recalculate a new set of random numbers for the table every time you load or view it. The numbers will constantly change as the expression is continually run to update the screen.

Active calculated columns are nice because they automatically update. However, the cost is the time it takes to calculate the value. If the source values are static and will never change, it is more efficient to add a static calculated column (see the static calculated recipe).

Note that the final example above (random number) depends upon the random library. Suppose you save the table and close Picalo. When you reopen Picalo, you must reimport the random library before loading the table. If you reload the table first, Picalo will not be able to find the *randint* function and will show an error in the cells.

1.18 Add a Static Calculated Column to a Table

Picalo GUI Recipe:

1. Double-click a table in the left-side project browser to view table data.
2. Select *File — Table Properties...* or click the 'wrench' icon on the toolbar.
3. The table properties dialog shows. Navigate above or below the column you want to add the new column by.
4. Click the 'star' icon on the left side of the dialog. One icon will add the new column above the current column; the other will add below.
5. Set the new column name. Set the column type to one of the types.
6. Add a calculation in the form of an expression. If you click the expression builder, a dialog will help you build the expression. Following are example expressions:
   - *Double the amount column*: `Amount * 2`
   - *Add two columns*: `UnitPrice + ItemTax`
   - *Use a Picalo function*: `Benfords.get_expected(Amount, 2)`
7. Ensure the "Active Calculation" box is unchecked.
8. Click the 'Save' button to finish.

Script Recipe:

```
# add a calculated column to the end of a table
data.append_calculated_static('Amount2', float, 'Amount * 2')

# add two columns; insert new col before Name column
data.insert_calculated_static('Name', 'Subtotal', float, 'UnitPrice + ItemTax')

# use a Picalo function; insert new col at the beginning of table
data.insert_calculated_static(0, 'Benford', float, 'Benfords.get_expected(Amount, 2)')

# insert a random number
import random
data.append_calculated_static('RandomNum', int, 'random.randint(1, 10)')
```

Discussion:

Static calculated columns record only the results of the expression you enter. The expression is used to calculate the new column values and then the expression is disposed of.

Static calculated columns are useful when you are analyzing static data (that will not change). They are much more efficient (i.e. faster) than active calculated columns because the expression is only evaluated one time. However, if the source values change, the calculated column values will not reflect the new changes.

For example, the final example (random number) will record the new random number, then it will never change. The value will be available on subsequent runs of Picalo, even if the random library is not loaded.

To repeat, use static calculated columns most of the time. Only use active calculations when you really need real-time updating.

1.19 Remove a Column From a Table

Picalo GUI Recipe:

1. Double-click a table in the left-side project browser to view table data.
2. Select *File — Table Properties...* or click the 'wrench' icon on the toolbar.
3. The table properties dialog shows. Navigate to the column you want to remove.
4. Click the 'x' button on the left side of the dialog. The column is removed.
5. Click the 'Save' button to finish.

Script Recipe:

```
1 # delete the Name column from the table
2 data.delete_column('Name')
3
4 # delete the Name column from the table (alternate syntax)
5 del data['Name']
6
7 # delete the third column in the table
8 data.delete_column(2)
9
10 # this deletes the third row, not the third column!
11 del data[2]
```

Discussion:

Columns can be deleted with either the *delete_column* method or the familiar *del* command. You should normally refer to columns by name, where both methods can be used.

If you refer to the column by numerical index, only the *delete_column* syntax can be used. Picalo always assumes that numerical indices in square brackets, such as [3], refer to records.

1.20 Copy an Entire Table

Picalo GUI Recipe:

1. Double-click a table in the left-side project browser to view table data.
2. Select *File — Copy Table...* .
3. Enter the new table name in the dialog that comes up.
4. The new table is now listed in the left-side browser.

Script Recipe:

```
1 # copy the data table to newdata
2 # this is the correct syntax
3 newdata = data[:]
4
5 # define a new variable to the existing table
6 # this does NOT copy the records (see discussion)
7 # and is probably the wrong syntax
8 newdata2 = data
```

Discussion:

Tables can be easily copied in Picalo. The new table is identical to the old one, but it is an entire new copy. Any changes made to the original will not be reflected in the new table.

Picalo has to load the entire table into memory to be able to copy it. You'll then have two copies of your data in memory at once. This might be prohibitive for large tables. If this is the case, you may want to simply save the table to disk and then copy the disk file using your operating system's file browser (e.g. Windows Explorer).

Copying a table in Picalo does not save the new table to disk. You need to explicitly copy the table to disk or you'll lose it when you close Picalo.

Programmers should note the difference between copying a table and simply defining a new variable. The first example above makes a new copy of the table and all its data. The second example (*newdata2 = data*) simply

defines a new variable to the existing data table. In the second example, any changes made to *newdata2* will also show in *data* because the two variables are pointing to a single table. It's as if you've given a nickname to the *data* table.

1.21 Copy Part of a Table

See the section earlier in this chapter named *Retrieve Several Table Records By Index*.

1.22 Combine Two Tables

Picalo GUI Recipe:

1. Not available.

Script Recipe:

```
# combine data1 and data2 to form data3
data3 = data1 + data2

# add the records of data2 to data1
data1.extend(data2)
```

Discussion:

Picalo supports the + operation between compatible tables. Tables are compatible when they have the same column names and types – in other words, the same structure. Picalo will create a third table that matches the structure of the two being added and add the records of both tables to the new table.

The second syntax, using *extend*, appends all of the records of the second table to the first table. In this case, a third table is not created. The second table remains unchanged and the first table includes the records of both tables.

1.23 Delete a Table

Picalo GUI Recipe:

1. Double-click a table in the left-side project browser to view table data.
2. Select *File — Remove Table From Project...*
3. A dialog comes up that allows you to do three things:
   (a) *Just remove the table from the project*: This option will remove the table from active memory (and free up memory for other tables). The project will also be removed from the project.
   (b) *Remove the table and delete the source file from disk*: This option will remove the table from active memory *and* delete the Picalo (.pco) file from disk, if the table has been saved. If the table has been exported to CSV, TSV, or another format, these files will not be removed.
   (c) *Do nothing*: This option is the same as canceling the operation.

Script Recipe:

```
# remove the data table from the project
del data

# just remove the data table's .pco file from disk
# the table remains in memory, and Picalo will
# recreate the disk file when data.save() is called
os.remove('c:/dir/data.pco')

# remove the data table from the project
# and remove its .pco file from disk
del data
os.remove('c:/dir/data.pco')
```

Discussion:

There are many ways to close and/or delete a table. Simply closing its tab (and leaving it in the left-side project browser) will remove it from memory and free up memory for other tables. This is normally sufficient.

If you really wan to get rid of the table, you can either just remove the table from your project or delete any associated .pco file from disk.

While it is possible to view a table from the shell or a script, it is not possible to close a table's tab without fully removing the table from the project.

The *os.remove* function is a direct Python function that will delete any file on your hard drive. We're using it here to remove a .pco file, but you could just as easily delete a delimited file with the function. Note that *os.remove* does not send the file to your operating system's trash or recycle bin – it just deletes the file permanently.

Chapter 2

Working With Table Lists

Many routines, such as stratification produce lists of tables rather than single tables. For example, suppose you have a list of employee records and you want to stratify by employee. If you have 15 employees in the data set, you'll end up with 15 new tables (one table per employee).

Picalo allows you to work with these lists of tables as if they were singular tables. You can use almost all the Picalo menu commands on them, and the functions run across the tables. In the employee example, if you add a new calculated column to the table list, the new column will be added to all 15 tables. If you ask for descriptives on the table list, you'll get a new table list of 15 summary descriptives – one per table in the original list.

Picalo defines two types of table lists: TableList and TableArray. The TableArray type is the more structured of the two. All tables in a TableArray list must have consistent table structures, meaning they have the same column names, types, and formats. Because the tables in a TableArray list have the same structure, Picalo allows you to use TableArray lists in place of regular Table objects in most routines. Most Picalo functions return TableArray lists.

The TableList type is simply a list of disjointed tables. It is the looser of the two list types in Picalo. TableLists are useful to hold a number of tables, but since the tables they contain are not necessarily similar, they cannot be used in most Picalo functions. Fortunately, they are not used very often – Picalo functions almost always return TableArray objects.

Since TableArrays are the more-powerful and most-used object, this chapter gives examples using TableArray.

2.1 Create a Table List

Picalo GUI Recipe: Many Picalo menu options, such as stratification, create table lists. You'll normally use these functions rather than create table lists manually. See the recipes and manual for Picalo functions for more information.

Script Recipe:

```
1 # assuming we have three tables, combine them to a table list
2 tlist = TableArray(data1, data2, data3)
3
4 # create a TableArray from a regular Python list
5 reglist = [ data1, data2, data3 ]
6 tlist2 = TableArray(reglist)
```

Discussion:

Normally, you'd let Picalo create TableArrays by using functions like *stratify_by_value)*. If you need to create TableArrays directly, use the TableArray constructor. In most ways, TableArray objects act like regular Python lists.

Placing tables into a table list does not copy the table data into new tables. It simply places the existing tables into a list. Any changes made to the list tables are made to the source tables – the two are one and the same.

2.2 Access an Individual Table in a List

Picalo GUI Recipe:

1. Double-click a table list object in the left-side project browser to view data.
2. Use the spinner at the top-right of the window to browser through the tables in the list.

Script Recipe:

```
1 # add a record to the first table in the list
2 tlist[0].append(1, 'Benny', 25000)
3
4 # print the name value from the fourth table, sixth record
5 print tlist[3][5].Name
```

Discussion:

Use the familiar list access idiom, the square brackets, to access tables in a list.

2.3 Add a Table to a Table List

Picalo GUI Recipe:

1. Not available.

Script Recipe:

```
# add a table to the end of a table list
tlist.append(data4)

# insert a table at the beginning of a table list
tlist.insert(0, data5)
```

Discussion:

Use the familiar list functions like insert and append to modify the tables in a table list.

2.4 Convert a Table List into a Table

Picalo GUI Recipe:

1. Right-click the table list in the left-side browser.
2. Select *Combine Into Regular Table* from the popup menu.

Script Recipe:

```
1 # assume we have a TableArray called tlist
2 newdata = tlist.combine()
```

Discussion:

Combining a table list into a single table creates a new table comprising all the records of the list. The original table list (and source tables) remains unchanged. Since the records are copied into a new table, you must have enough memory to hold the new table.

Chapter 3

Basic Table Analysis

This chapter presents recipes for working with tables once they are loaded. The routines are basic and include actions like sorting, totaling, filtering, and so forth.

3.1 Making a Table Read-Only

Picalo GUI Recipe:

1. Double-click a table in the left-side project browser to view table data.
2. Click the read-only button at the top of the table (just to the left of the Filter area).

Script Recipe:

```
1 # set read only
2 data.set_readonly(True)
3
4 # make editable again
5 data.set_readonly(False)
```

Discussion:

When analyzing tables, it is often important to set them as read-only so you don't accidentally change their data. By simply clicking the read-only button, you can ensure that tables are not modified during analysis.

Database tables (retrieved from a database connection) are always read-only and cannot be modified. Use INSERT and UPDATE queries to modify database tables.

3.2 View Table Descriptives

Picalo GUI Recipe:

1. Double-click a table in the left-side project browser to view table data.
2. Select *Analyze — Descriptives...* .
3. A new table shows containing the descriptives for the table.

Script Recipe:

```
1 data_descriptives = Simple.describe(data)
2 data_descriptives.view()
```

Discussion:

Descriptives are an important first step you should always complete when analyzing a table. Descriptives show basic statistics, control totals, and record counts. You should use these types of descriptives to ensure that your data imported correctly, were typed to the correct type without problems, and no records were missed.

Most of the descriptives are self-explanatory. The *NumEmpty*, *NumNone*, and *NumZero* are subtly different. The *NumEmpty* descriptive tells you how many cells are the empty string (""). The *NumNone* descriptive tells you how many cells are set to the None type, regardless of the column type (since any cell anywhere in a table can be set to None). The *NumZero* descriptive tells you how many cells are set to 0 (for numerical typed columns).

3.3 Validate Column Data

Picalo GUI Recipe:

1. Double-click a table in the left-side project browser to view table data.
2. Click the "magic wand" icon in the Filter Expression area (just above the table).
3. Type a pattern using either wildcard syntax or regular expression syntax. Example patterns are in the discussion area below.
4. Check the *Exclude Matching Records* radio button. This will filter out all records that match the pattern, showing records that do not validate.
5. Repeat the process for each column you wish to validate.

Script Recipe:

```
1 # add a filter to the table
2 data.filter('not Simple.wildcard_match("#?*"', ColName)
```

Discussion:

One of the most common tasks after loading data into Picalo is to validate that all data in a given column follows a certain pattern. For example, you may want to leave all columns as string types when importing delimited files (assigning column types directly in the wizard may throw errors on values that can't be converted correctly). Then use Picalo's filtering functions to validate the values in each column, such as email addresses, numbers, date formats, or other validations. Once validated, you can then go to Table Properties and assign the correct column types with confidence. This pattern works well when working with poorly-formatted data or with files that have known errors in them.

Picalo provides two methods of pattern matching: wildcards and regular expressions. Wildcards are easy to create and are less prone to errors. Regular expressions are more powerful but require understanding of a complex pattern matching language. Picalo's *Filter Table* dialog, as shown in the GUI recipe above, makes filtering easy.

Wildcard patterns support only three special characters: #, ?, and *. All other characters are matched literally. The special characters are as follows:

- A pound sign (#) matches a single number: 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9.
- A question mark (?) matches a single letter: A-Z or a-z. Regardless of the ignorecase flag in the function, the question mark always matches any letter (upper or lower).
- A star (*) matches zero or more letters (?) or numbers (#). It matches an empty string, 'aaa', 'a903j', and 'A523BC'. It tries to match as many characters as it can, until it hits a non-letter and non-number character like a space, punctuation mark, or any other special character.

Wildcards are limited to keep them simple and easy. With wildcards, you cannot match a literal pound sign (since it is always interpreted as a placeholder for a number), use a hard return (since we have no character on the keyboard for it), or employ boolean logic (either-or situations). If you need these capabilities, it's time to step up to full regular expressions.

The following are example wildcard patterns:

- `1-###-###-####` matches a US or Canada phone number. The field must start with a literal 1-, then three numbers, another -, three numbers, another -, and four numbers.
- `?*@?*.??*` is a simple way to match an email address. It looks for a letter followed by other letters or numbers, then an @ sign, then a letter followed by other letters or numbers, then two letters followed by other letters or numbers (for the country code or top-level domain). Note that usernames with a dot in them (like homer.simpson@tv.com) or that start with a number (like 5hom@tv.com) will not match this pattern.

Regular expressions are one of the oldest and most widely-used pattern matching languages in computers today. Almost all modern computer languages, and many applications, support matching via regular expression patterns. Picalo uses Python's built-in regular expression engine, so you can have confidence that your patterns will be interpreted correctly and in a standard way.

The syntax of regular expressions is beyond this discussion. Readers are encouraged to search the Internet for information on and examples of regular expression. Python's particular dialect is described at `http://docs.python.org/dev/howto/regex.html`.

In addition to *Simple.regex_match*, Python's standard `re` module is available in all Picalo expressions and scripts. While *Simple.regex_match* returns only a True/False match answer, the full `re` module allows group matching, look aheads and look behinds, field substitution, and many more features.

The following are examples of regular expression patterns:

```
1 ^.+@.+\.\w{2,4}$
```

The above example is a simple regex to match email addresses. Much more powerful patterns for email addresses can be found online.

```
1 1-\d{3}-\d{3}-\d{4}
```

This example matches a US or Canada phone number. The `d{3}` specifies three numbers.

```
1 ^P(ost){0,1}\.{0,1} *O(ffice){0,1}\.{0,1} *Box$
```

This is a great example of the power of regular expressions because a single pattern can match so many different (but valid) values. It matches P.O. Box, P. O. Box, Post O. Box, post office box, post. office. box, and many other variations of the words.

3.4 Total a Column

Picalo GUI Recipe:

1. Double-click a table in the left-side project browser to view table data.
2. Select *Analyze — Descriptives...* .
3. A new table shows containing the descriptives for the table.
4. The *Total* descriptive gives you the total of each column.

Script Recipe:

```
1 # sum the Amount column
2 total = sum(data['Amount'])
3 print total
```

Discussion:

Totaling a column (or doing any summary calculation on a column) uses the table['colname'] syntax to retrieve a column. Since columns are like Python lists, you can use any list-oriented function on them. Common functions include sum, mean, stdev, variance, max, and min.

3.5 Analyze Each Record in a Table

Picalo GUI Recipe:

1. Not available.

Script Recipe:

```
# iterate across each record
for rec in data:
  rec['Amount'] = round(rec['Amount'], 2)

# iterate across each record with the record index
for i, rec in enumerate(data):
  rec['RecNo'] = i

# iterate with just an index
for i in range(len(data)):
  table[i]['Amount'] = round(table[i]['Amount'], 2)
```

Discussion:

Use a *for* loop to iterate across an entire table, one record at a time. The code will run for each record in the table, with the *rec* variable set to each record as the iteration progresses. The first example rounds each value in the Amount column to two decimal places.

Normally, you don't need to know the index of each record (i.e. record number) as you iterate. Just getting a reference to the actual record is enough to accomplish your task.

If you need to know each record number, use the *enumerate* function (second example) to set *i* to the index as you iterate.

If you only need the record number, use the familiar Python *range* and *len* functions to go through the entire table. This method does not give you a reference to each record – you need to use the table variable.

3.6 Search a Table

Picalo GUI Recipe:

1. Double-click a table in the left-side project browser to view table data.
2. Select *Edit — Find and Replace...* .
3. Enter the text to search for and click the Find button.

Script Recipe:

```
# search for all cells with of AMC Corp
for rec in data:
    for cell in rec:
        if cell == 'AMC Corp':
            print 'Found AMC'

# search within cells for "AMC"
for rec in data:
    for cell in rec:
        if 'AMC' in str(cell):
            print 'Found AMC somewhere in the cell'

# search within cells for "AMC" (case insensitive)
for rec in data:
    for cell in rec:
        if 'amc' in str(cell).lower():
            print 'Found AMC somewhere in the cell'
```

Discussion:

The Picalo GUI supports direct find and replace within an open table. It has options for starting at the top of the table, matching within cells or entire cells, and case sensitivity.

In code, search and replace requires a double for loop to first loop through records and then through the cells in each record. The examples above show how to search for entire and partial cell matches. Case insensitivity is easy to do with the *lower* function, which will convert any cell text into lowercase.

Picalo has more advanced ways of searching as well. In fact, Picalo itself is primarily a searching application. These are described in other recipes.

- Discovery of duplicate values in a column.
- Discovery of gaps in sequence.
- Selection of records based upon an exact value or an arbitrary expression. Selection by expression is limited only by your imagination.
- Identification of outliers using z-score or other methods.
- Finding of matching or non-matching records between tables.
- Use of database queries to find records.
- Fuzzy matching of strings.
- Wildcard matching using #, ?, and *.
- Regular expressions, the ultimate in pattern and mask searching.

3.7 Filter a Table

Picalo GUI Recipe:

1. Double-click a table in the left-side project browser to view table data.
2. Enter an expression in the Filter box at the top of the table and click the Set button. The 'magic wand' button opens the expression builder to help you build an expression. Example expressions are as follows:
   - InvoiceNumber == 15
   - InvoiceNumber ¿ 15 and InvoiceNumber ¡ 50
   - Name == 'AMC Corp'
   - 'amc' in Name.lower()

Script Recipe:

```
1 data.filter("InvoiceNumber == 15")
2 data.filter("InvoiceNumber > 15 and InvoiceNumber < 50")
3 data.filter("Name == 'AMC Corp'")
4 data.filter("'amc' in Name.lower()")
```

Discussion:

Filtering is an essential function for analyzing data in tables. Filtering removes all records (rows) from a table that are not pertinent to your analysis. The removed records are not actually deleted, but only temporarily hidden. Hidden records are excluded from almost all functions in Picalo – as if they were not in the table at all. These records can be restored at anytime. You should read the tutorial on filtering in the introductory manual for a detailed introduction to this topic.

Expressions must be valid Python expressions that evaluate to True or False. Records that evaluate to True are shown, those evaluating to False are hidden. Remember that the operator for equality testing is the double equals sign (==), not the single equals sign (=).

The record index of the table records will change when a table is filtered. In other words, the visible records will still be indexed as 0, 1, 2, 3, 4, etc. If you had 10 records before filtering (and filtering made 5 of them hidden), you now have indices 0, 1, 2, 3, and 4, and the length of the table is temporarily 5. Releasing the filter will restore the original indices and table length.

Filtering is very similar to selecting by expression. Both use expressions to select matching records. The difference is selecting creates a new table of matching records and leaves the existing table unchanged. Filtering just hides nonmatching records within the existing table. Filtering is more efficient because it doesn't make a copy of the records.

3.8 Filter a Table Using Wildcards

Picalo GUI Recipe:

1. Double-click a table in the left-side project browser to view table data.
2. Enter an expression in the Filter box at the top of the table and click the Set button. Use one of several available functions to enable wildcard matching. Example expressions are as follows:
   - 'Maple' in NameCol
   - 'maple' in NameCol.lower()
   - NameCol.find('Maple') ¿=0 and NameCol.find('Maple')¡ 5
   - Simple.fuzzymatch('Maple', NameCol) ¿ 0.3
   - Simple.wildcard_match('Maple *', NameCol')
   - Simple.regex_match('.*Maple +Street', NameCol)

Script Recipe:

```
1 data.filter("'Maple' in NameCol")
2 data.filter("'maple' in NameCol.lower()")
3 data.filter("NameCol.find('Maple') >=0 and NameCol.find('Maple')< 5")
4 data.filter("Simple.fuzzymatch('Maple', NameCol) > 0.3")
5 data.filter("Simple.wildcard\_match('Maple *', NameCol')")
6 data.filter("Simple.regex\_match('.*Maple +Street', NameCol)")
```

Discussion:

Picalo tables can be filtered with wildcards in several ways. Readers should first read the previous recipe, *Filtering a Table* to learn basic filtering techniques. This recipe presents more advanced concepts related to filtering string-typed columns with wildcards.

The formulas used in the above examples are described as follows:

- `'Maple' in NameCol`: This is probably the most simple way of looking for inexact matches. The formula uses Python's native *in* keyword, which returns true if the left side ('Maple') is found anywhere in the right side (NameCol).

- `'maple' in NameCol.lower()`: A variation on the last formula, this expression uses the string *lower* method to convert the value in NameCol to lowercase before comparing. Since 'maple' is also given in lowercase, the effect is a case-insensitive match.

- `NameCol.find('Maple') >=0 and NameCol.find('Maple')< 5`: This expression uses the string *find* method to get a more specific result than the previous formulas. The *find* method returns the index that 'Maple' starts on. For example, if a cell contained '30 Maple Street', *find* returns 3, indicating that the words starts in the third index (remember, zero-based). If the word is not found anywhere in the cell, the method returns -1 (hence the ¿=0 part).

- `Simple.fuzzymatch('Maple', NameCol) > 0.3`: This expression uses Picalo's *fuzzymatch* function to get an approximate match. The function returns a value between 0 and 100 percent; the example is looking for approximate matches greater than 30 percent. See the Simple.fuzzymatch documentation for more information on this excellent function.

- `Simple.wildcard_match('Maple *', NameCol')`: This expression uses Picalo's wildcard terms to find the word Maple, then a single space, then any other word. See Recipe 3.3 for more information on this type of wildcard matching.

- `Simple.regex_match('.*Maple +Street', NameCol)`: This expression is the most advanced and powerful method of searching. It uses Python's regular expression library. Regular expressions are one of the most useful pattern matching languages in existence (they are available in most modern computer languages). This example looks for some text, followed by 'Maple', then one or more spaces, then 'Street'. See Recipe 3.3 for more information on this type of wildcard matching.

Note that these formulas can also be used in the *Data — Select — By Expression* dialog.

3.9 Clear a Filter from a Table

Picalo GUI Recipe:

1. Double-click a table in the left-side project browser to view table data.
2. Click the Release button to release the filter.

Script Recipe:

```
1 data.clear_filter()
```

Discussion:

Clearing a filter removes the filter expression from a table. All records in the table are restored to the table. The original record indices and table length are restored.

3.10 Sort a Table

Picalo GUI Recipe:

1. Double-click a table in the left-side project browser to view table data.
2. Select *File — Sort...* .
3. Select the fields to sort by in the dialog that comes up.

Script Recipe:

```
1 # sort data (ascending) by InvoiceNum, then Date
2 Simple.sort(data, True, 'InvoiceNum', 'Date')
3
4 # sort data (descending) by InvoiceNum, then Date
5 Simple.sort(data, False, 'InvoiceNum', 'Date')
```

Discussion:

The Picalo GUI supports sorting a table by up to three fields. The script language supports sorting by any number of fields.

The Simple.sort method requires that all fields be sorted ascending or descending. For increased flexibility, use the *table.sort* method to sort a table using the standard Python sort idiom. See the Python sorting mini-tutorial for more information on this type of advanced sorting.

Chapter 4

Loading and Saving Data

Picalo supports loading and saving data in many formats. The primary Picalo format, .pco, saves all information about a table, including column names, types, etc. Picalo format is suitable for small- and medium-sized tables.

The best practices method of keeping data in Picalo is to connect it to a data warehouse using MySQL, PostgreSQL, or another ODBC data source. This allows the database to do what it was programmed to do (store data) and Picalo to do what it was primarily programmed to do (analyze data).

Picalo supports importing and exporting of delimited text files (.tsv, .csv, .txt), fixed width text files (usually .txt), and classic Excel files (.xls).

4.1 Load a Picalo Table

Picalo GUI Recipe:

1. Select *File — Open Table...* .
2. Navigate to the table you want to open and click 'Open'.

Script Recipe:

```
1 data = load("c:/MyFiles/data.pco")
```

Discussion:

The primary Picalo format, .pco, contains all information about a table, including column names, types, etc. Picalo format is suitable for small- and medium-sized tables. If you have an extremely large table, use a database.

4.2 Import a Delimited Text File

Picalo GUI Recipe:

1. Select *Data — Data Import Wizard...*
2. Follow the wizard instructions for the type of file you are importing.

Script Recipe:

```
# load a tab-separated file
data = load_tsv('c:/MyFiles/data.tsv', header_row=True)
data.set_type('Amount', float)
data.set_type('BirthDate', Date)

# load a comma-separated file
data2 = load_tsv('c:/MyFiles/data.csv', header_row=True)

# load a custom delimited-text file
data3 = load_delimited('c:/MyFiles/data.txt', header_row=True, delimiter='~', qualifier='"')
```

Discussion:

Delimited text files are one of the most common file formats used to transfer data between applications. If the application you are retrieving data from can output tab-separated, comma-separated, or other delimited files, Picalo will have no problem importing the file.

Delimited text files use a delimiter character, such as a comma or a tab, to separate the fields in each row. The first row of text is usually the field headers, but sometimes the headers are not included. Be sure to get the data dictionary (also called schema) from the source application so you know the fields you are getting.

Since delimited text files do not include column types, Picalo assumes all data are string/unicode values. The data import wizard will help you set the column types to the right types. Scripts need to call *set_type* after loading the delimited file.

4.3 Import a Fixed Width Text File

Picalo GUI Recipe:

1. Select *Data — Data Import Wizard...*
2. Follow the wizard instructions for the type of file you are importing. The wizard will allow you to click on the table to define where columns start and finish.

Script Recipe:

```
1 # load a fixed-width file with four columns
2 data = load_fixed('c:/MyFiles/data.tsv', [0, 12, 19], header_row=True)
3 data.set_type('Amount', float)
4 data.set_type('BirthDate', Date)
```

Discussion:

Fixed width files use white space to separate the fields in each row. Fixed width files are commonly used in legacy applications, especially in applications built in the 1970's and 1980's. Fields start and end at exact character positions, such as position 0, 12, and 19. Use a list of numbers to define the starting location of each column. The wizard lets you define these positions by simply clicking the mouse on an a sample from your data.

The first row of text is usually the field headers, but sometimes the headers are not included. Be sure to get the data dictionary (also called schema) from the source application so you know the fields you are getting.

Since fixed width text files do not include column types, Picalo assumes all data are string/unicode values. The data import wizard will help you set the column types to the right types. Scripts need to call *set_type* after loading the delimited file.

4.4 Import an EBCDIC Data File

Picalo GUI Recipe:

1. Not available.

Script Recipe:

```
# load the data into a new table, specifying column positions as 0-2, 2-7, and 7-13
data = load_fixed('test.txt', [0,2,7,13], encoding='cp037', line_separators=False)
data.set_type('id', int)
data.set_type('salary', number)
```

Discussion:

Extended Binary Coded Decimal Interchange Code (EBCDIC) is an 8-bit character encoding created by IBM for use on its mainframes. It is a widely despised character encoding, but it is often the only available export from older systems. One of the primary reasons programmers dislike it is it matches character code numbers to the old punch card locations.

It is important to understand exactly what EBCDIC is and what it is not. It is *not* a data structure like comma separated values or XML. EBCDIC is merely a matching of keyboard characters to code numbers for representation inside the computer. For example, the letter 'a' is hex code 61, 'b' is hex code 62, and so forth. EBCDIC is an alternative to the more popular ASCII or UTF-8 encodings.

Picalo can handle any number of encodings, of which EBCDIC is only one. Other popular encodings are latin_1, iso8559_2, utf-8, utf-16, mac_roman, and of course, ascii. The available codecs are listed in the codecs module at:

`http://docs.python.org/lib/standard-encodings.html`

EBCDIC is encoding 'cp037' in Picalo, which specifies the English version of EBCDIC.

Because EBCDIC simply defines the character numbers, the table data can actually be any structure. It could be stored as comma-separated values, fixed format, or even XML! It is impossible for Picalo to guess what export structure an application used.

Fortunately, *many* mainframes export data in fixed-width format without line separators. It is this structure that the recipe loads. 'Without line separators' means that the data file simply runs one line of data onto the end of the ones before it, without any hard returns (enter key characters). This makes it a little more difficult to load because you have to exactly specify the column start and end positions, but Picalo can load it just fine. The `line_separators=False` parameter tells the `load_fixed` function to not look for hard returns in the file. The following example shows a file with three records, all run into a single line:

```
idname salary01Marge05000002Dan   004500
```

The actual table looks like this:

```
+----+--------+--------+
| id |   name | salary |
+----+--------+--------+
| 01 | Marge  | 050000 |
| 02 | Dan    | 004500 |
+----+--------+--------+
```

The column positions, given as `[0, 2, 7,13]`, specify three columns. The notation is inclusive of the starting character and exclusive of the ending character.

4.5 Import an XML Data File

Picalo GUI Recipe:

1. Not available.

Script Recipe:

```
1 import xml.dom.minidom
2 document = xml.dom.minidom.parse('c:/temp/mydata.xml')
3 root_element = document.documentElement
4 # now that we have the root element, parse the dom tree for data
```

Discussion:

Python's xml libraries can import any type of xml file, although you have to write the code manually to import the data and populate a Picalo table. See the standard documentation for the Python xml libraries for more information.

4.6 Import a Microsoft Excel File

Picalo GUI Recipe:

1. Select *Data — Data Import Wizard...*
2. Follow the wizard instructions for Excel files. It will allow you to enter the starting and ending cells to import.

Script Recipe:

```
1 data = load_excel('c:/MyFiles/data.xls', 'Sheet1', 'A1', 'C200', header_row=True)
2 data.set_type('Amount', float)
3 data.set_type('BirthDate', Date)
```

Discussion:

Picalo can read native Excel files. Since spreadsheets don't always start with the first cell (and don't always contain the data you want on the first sheet), the method allows you to enter the sheet name as well as the cell range to import.

Since Excel columns are not explicitly typed, Picalo initializes the table with string/unicode column types. After you import the data, set column types to their correct types.

4.7 Save a Picalo Table

Picalo GUI Recipe:

1. Double-click a table in the left-side project browser to view table data.
2. Select *File — Save Table* . Enter the filename and click 'Save'.

Script Recipe:

```
1 # save all records
2 data.save('c:/temp/myfile.pco')
3
4 # save only records matching the current filter
5 data.save('c:/temp/myfile.pco', respect_filter=True)
```

Discussion:

The primary Picalo format, .pco, contains all information about a table, including column names, types, etc. Picalo format is suitable for small- and medium-sized tables. While there is no inherent limit to the number of records you can save into a .pco file, using a database for large data sets is more efficient.

If the table is filtered (and some records are hidden), the method ignores the filter and saves all records. If you want to save only the filtered records, use the format of the second example.

4.8 Export a Delimited Text File

Picalo GUI Recipe:

1. Double-click a table in the left-side project browser to view table data.
2. Select *File — Export Table...* . Select the appropriate file type (CSV, TSV, etc.) and click 'Save'.

Script Recipe:

```
# save a tab-separated file
data.save_tsv('c:/mydata/mytable.tsv')

# save a comma-separated file
data.save_csv('c:/mydata/mytable.csv')

# save a custom delimited file
data.save_delimited('c:/mydata/mytable.csv', delimiter='~')
```

Discussion:

Delimited text is one of the most common transfer formats available today. It is understood by almost any data-enabled application, including most databases.

The Picalo functions allow you to set the delimiter, qualifier, line ending (Unix or Windows file endings), and encoding (for international files). In addition, you can indicate whether you want any table filters respected or not (see recipe 4.7).

The method will automatically take care of delimiters or qualifiers in the actual field data. For example, if you have a comma in one of your field values, you can still use CSV format. The use of special characters, such as a tilde because you don't expect a tilde in your data, is unnecessary.

Delimited text files do not contain any data typing information – all data will be seen as strings in the output file. You will lose any column expressions and filters. Use delimited text format only when you need to export to another application.

4.9 Export a Fixed Width File

Picalo GUI Recipe:

1. Not available

Script Recipe:

```
1 data.save_fixed('c:/mydata/mytable.tsv')
```

Discussion:

Fixed width files use white space to separate the fields in each row. Fixed width files are commonly used in legacy applications, especially in applications built in the 1970's and 1980's. Fields start and end at exact character positions, such as position 0, 12, and 19. Picalo will automatically determine the column positions based on the data in your table. The method parameters allow you to indicate whether you want any table filters respected or not (see recipe 4.7).

Fixed width text files do not contain any data typing information – all data will be seen as strings in the output file. You will lose any column expressions and filters. Use fixed text format only when you need to export to another application. If the application you are exporting to can read both delimited text and fixed width text files, I recommend delimited text files as they are more explicit and more widely used.

4.10 Export an XML Data File

Picalo GUI Recipe:

1. Not available.

Script Recipe:

```
1 data.save_xml('c:/mydata/mytable.xml')
```

Discussion:

Picalo can export tables to a specific xml format. This is a one-way transfer; Picalo does not automatically read the xml file back into a table (you'd have to write that code yourself using Python's xml libraries). The method saves as much information as possible, including column types and formats.

The xml saving functionality is included to allow export to applications that only take xml. The method signature allows definition of line endings (Unix or Windows), the indent character for prettier xml, a compact flag for more compact xml, and other options.

4.11 Export a Microsoft Excel File

Picalo GUI Recipe:

1. Double-click a table in the left-side project browser to view table data.
2. Select *File — Export Table...* . Select the the Excel file type and click 'Save'.

Script Recipe:

```
1 data.save_excel('c:/myfiles/mydata.xls')
```

Discussion:

Picalo can save to the native Excel .xls format. It saves your data in Sheet1 starting with cell A1. All column types and formats will be lost in the process. You can indicate whether you want any table filters respected or not (see recipe 4.7).

Chapter 5

Working With Databases

The best practices method of keeping data in Picalo is to connect it to a data warehouse using MySQL, PostgreSQL, or another ODBC data source. This allows the database to do what it was programmed to do (store data) and Picalo to do what it was primarily programmed to do (analyze data).

Picalo provides access to relational databases through the Database module. It comes with the drivers for three types of connections: ODBC, PostgreSQL, and MySQL. The ODBC driver can connect to any type of database you can set up an ODBC connection for.

5.1 Connect to a Database

Picalo GUI Recipe:

1. Select *Data — Connect to Database...* .
2. In the dialog that comes up, select your type of database, and fill out the fields for your settings.
3. Click OK. Your database connection is now ready for use by the other options under the *Data* menu.

Script Recipe:

```
# use an ODBC connection with dsn 'mydb'
conn = Database.OdbcConnection('mydb', username='me', password='ps')
conn.close()

# create a PostgreSQL connection to 'mydb'
conn2 = Database.PostgreSQL('mydb', username='me', password='ps')
conn2.close()

# create a PostgreSQL connection to 'yourdb' on a different computer
conn3 = Database.PostgreSQL('yourdb', username='me', password='ps', host='10.0.5.1')
conn3.close()

# create a MySQL connection to 'mydb'
conn4 = Database.MySQLConnection('mydb', username='me', password='ps')
conn4.close()
```

Discussion:

Picalo includes the drivers for three types of databases: ODBC, PostgreSQL, and MySQL. ODBC connections allow you to connect to almost any relational database in existence. Picalo's ODBC connection requires that you install and set up an ODBC driver in your operating system. In Windows, first install the ODBC driver for your database. ODBC drivers can be downloaded for free from most database vendor sites. Once the driver is installed, go to Control Panel — Administrative Tools — ODBC and set up the connection.

Picalo's PostgreSQL and MySQL drivers allow direct connections to these databases (without the need to set up ODBC connections). Both of these databases are free to use, and both are excellent options for all but the very largest installations.

Note that with all Picalo connections, transactions are used for data changes. You must call *conn.commit()* to make your changes permanent in the database.

5.2 View Database Tables

Picalo GUI Recipe:

1. Connect to a database so it is listed in the left-side object browser.
2. Click the little plus symbol next to the database name to expand the tree item. The tables in the database will be listed.
3. For efficiency reasons, Picalo does not automatically refresh the list of tables. If new tables are created on the database server, right-click the database connection and select *Refresh* .

Script Recipe:

```
1 # list the tables in mydb
2 conn = Database.OdbcConnection('mydb', username='me', password='ps')
3 for tablename in conn.list_tables():
4     print tablename
5 conn.close()
```

Discussion:

Picalo can read the public tables from the database. Each time you call list_tables() (or select *Refresh* from the popup menu), Picalo queries the database to get a fresh list of tables.

5.3 Run an SQL Query

Picalo GUI Recipe:

1. Connect to a database so it is listed in the left-side object browser.
2. Select *Data — Run Visual Query...* .
3. The *Visual Query* dialog helps you build an SQL statement. You can also type the SQL directly in the text box at the bottom of the dialog.

Script Recipe:

```
1 conn = Database.OdbcConnection('mydb', username='me', password='ps')
2 data = conn.table("SELECT * FROM mytable")
3 # do something with table
4 conn.close()
```

Discussion:

One of Picalo's greatest strengths is its ability to query databases directly. Result records are pulled from the database in a just-in-time format, so even the largest queries should come back from the database as soon as the database has results.

The *table()* method (used by the Visual Query dialog as well) is normally the best way to query a database. Picalo takes the results returned from the query and creates a Picalo table, with column names, types, etc. These query-based tables can then be used in all Picalo dialogs and functions.

Query-based tables are always read-only. If you want to change the source data, use an INSERT or UPDATE query. It is not possible to change the read-only status of these tables. If you need to change the data in the table (but not in the database), copy the query-based table to a regular Picalo table (see Recipe 1.20).

If you have an extraordinarily large results set that cannot be contained in memory, iterate over the result set (see Recipe 5.4).

5.4 Run an SQL Query Efficiently

Picalo GUI Recipe:

1. Not available.

Script Recipe:

```
conn = Database.OdbcConnection('mydb', username='me', password='ps')
for rec in conn.query("SELECT * FROM mytable"):
  print "The name is:", rec['Name']
conn.close()
```

Discussion:

This recipe is very similar to the previous one on querying records into a Picalo table (Recipe 5.3), but it is much more efficient. It works even if you have hundreds of millions of records in your database. Generally, you should use the previous recipe, using the *table* function rather than *query*. The *table* function allows you to skip around in the records, go forwards and backwards, further filter the results, and use the results in Picalo functions.

The subtle difference in the two methods is the use of *conn.query* in this recipe. This method allows only iteration over the result set, one record at a time, as shown in the example. In other words, you must go through the records one at a time, from start to finish. No going backwards, skipping around, etc.

The reason this method is so efficient is Picalo only loads one record at a time from the database, so it uses almost no system resources on the Picalo side.

5.5 Insert a Record into a Database

Picalo GUI Recipe:

1. Not available.

Script Recipe:

```
# insert a record via standard SQL
conn = Database.OdbcConnection('mydb', username='me', password='ps')
id = 15
nm = 'Barry'
conn.execute("INSERT INTO mytable (id, name) VALUES (%s, %s)', (id, nm)")
conn.commit()
conn.close()

# insert a record via string concatenation (less effective method)
conn = Database.OdbcConnection('mydb', username='me', password='ps')
id = 15
nm = 'Barry'
conn.execute("INSERT INTO mytable (id, name) VALUES (" + id + ", '" + nm + "')")
conn.commit()
conn.close()

# insert a record via a query builder
conn = Database.OdbcConnection('mydb', username='me', password='ps')
id = 15
nm = 'Barry'
qb = conn.insert_query_builder('mytable')
qb.add('id', id)
qb.add('name', nm)
qb.execute()
conn.commit()
conn.close()
```

Discussion:

The first example of inserting records in this recipe is the most succinct. It uses the standard Python driver method of writing the SQL statement yourself. Field values are denoted with a percent sign in the first parameter, the SQL. The second parameter is a tuple (or list) of values to be inserted into the SQL statement upon execution. This is the preferred way of constructing SQL statements manually because it allows the driver to handle special

characters (like commas or quotes) automatically. Note that the second %s in the string does not require single quotes, even though the *nm* variable is a string.

The second example in this recipe uses string concatenation to build the query string. This method is generally a poor way to construct SQL because it is fragile. If quote or comma characters appear in the values (like *nm* above), the SQL string will not execute. In addition, this method is subject to SQL injections and other security problems. The first example solves all of these problems automatically by allowing the driver to lay the values into the SQL.

The third example shows Picalo's query builder feature. This feature allows more robust creation of SQL and more readable code. The method builds the SQL piece by piece rather than all in one statement.

Note that with all the above methods, you must call *commit* to make your changes permanent. If you forget this step, all your changes will be lost when you close the connection.

5.6 Update a Record in a Database

Picalo GUI Recipe:

1. Not available.

Script Recipe:

```
# update a record via standard SQL
conn = Database.OdbcConnection('mydb', username='me', password='ps')
id = 15
nm = 'Danny'
conn.execute("UPDATE mytable SET name=%s WHERE id=%s", (nm, id))
conn.commit()
conn.close()

# update a record via string concatenation (less effective method)
conn = Database.OdbcConnection('mydb', username='me', password='ps')
id = 15
nm = 'Danny'
conn.execute("UPDATE mytable SET name='" + nm + "' WHERE id=" + id)
conn.commit()
conn.close()

# insert a record via a query builder
conn = Database.OdbcConnection('mydb', username='me', password='ps')
id = 15
nm = 'Danny'
qb = conn.update_query_builder('mytable')
qb.add('name', nm)
qb.add_where('id', id)
qb.execute()
conn.commit()
conn.close()
```

Discussion:

The first example of updating records in this recipe is the most succinct. It uses the standard Python driver method of writing the SQL statement yourself. Field values are denoted with a percent sign in the first parameter, the SQL. The second parameter is a tuple (or list) of values to be inserted into the SQL statement upon execution. This is the preferred way of constructing SQL statements manually because it allows the driver to handle special

characters (like commas or quotes) automatically. Note that the second %s in the string does not require single quotes, even though the *nm* variable is a string.

The second example in this recipe uses string concatenation to build the query string. This method is generally a poor way to construct SQL because it is fragile. If quote or comma characters appear in the values (like *nm* above), the SQL string will not execute. In addition, this method is subject to SQL injections and other security problems. The first example solves all of these problems automatically by allowing the driver to lay the values into the SQL.

The third example shows Picalo's query builder feature. This feature allows more robust creation of SQL and more readable code. The method builds the SQL piece by piece rather than all in one statement.

Note that with all the above methods, you must call *commit* to make your changes permanent. If you forget this step, all your changes will be lost when you close the connection.

5.7 Upload an Entire Table to a Database

Picalo GUI Recipe:

1. Connect to a database so it is listed in the left-side object browser.
2. Create or load a Picalo table so it is listed in hte left-side object browser.
3. Select *Data — Upload Table to Database...*
4. In the dialog that comes up, select your table name and database connection name. Enter the new database relation name (i.e. the table that will be created).
5. Click OK to upload the table.

Script Recipe:

```
1 # load a Picalo table
2 data = load_tsv('c:/MyFiles/data.tsv', header_row=True)
3 data.set_type('Amount', float)
4 data.set_type('BirthDate', Date)
5
6 # connect and upload to the database
7 conn = Database.OdbcConnection('mydb', username='me', password='ps')
8 conn.post_table(data, "employees", replace=False)
9 conn.close()
```

Discussion:

Picalo contains an experimental feature to upload entire Picalo tables to database relations. This feature is experimental because subtle differences between databases may cause the method to fail. Please test the method on your database before trusting it to real tables and databases.

To upload the table, Picalo first inspects the table for column names and types. It constructs a CREATE TABLE statement for the table and executes it. The *replace* parameter allows you to control whether existing relations in the database are overwritten. Picalo then executes INSERT queries for each record in the table to upload the data to the database.

Note the absence of the *commit* statement in the example code. For several reasons, the *post_table* method must commit after each record is uploaded. Therefore, there is no need to manually commit after the upload command.

The method can be a very useful way to import delimited text files (or other sources) into any database. While some DBMS front ends allow you to import data, Picalo can help you insert records when importing is not available. This function can also query a table from one database and post the data to another.

5.8 Copy a Table From One Database to Another

This can be done automatically with Recipe *UploadTable*.

5.9 Create a Database Index

Picalo GUI Recipe:

1. Not available in Picalo. Please consult your database documentation for graphical ways to create indices.

Script Recipe:

```
1 conn = Database.OdbcConnection('mydb', username='me', password='ps')
2 conn.execute("CREATE INDEX mytableindex1 ON mytable(id, name)")
3 conn.close()
```

Discussion:

Picalo automatically creates indices as needed for its own tables. You should never need to manually create indices on Picalo tables.

However, databases require that you create indices on database relations to speed up queries. For large data sets, database indices can mean the difference between a 30 minute query and a 24 hour query. While the concept of database indices are beyond the scope of this recipe, you should always examine your SQL queries to see what fields you are filtering on (i.e. the WHERE clauses). Adding indices for these fields will significantly speed up your queries.

The drawback to indices is the additional overhead required for the database to maintain each index. Indices must be updated each time you insert, update, or delete a record. Indices also use additional disk space and memory.

5.10 Delete a Record From a Database

Picalo GUI Recipe:

1. Not available.

Script Recipe:

```
1 conn = Database.OdbcConnection('mydb', username='me', password='ps')
2 id = 15
3 conn.execute("DELETE FROM mytable WHERE id=%2", (id, ))
4 conn.commit()
5 conn.close()
```

Discussion:

In the above example, all records with id equal to 15 are deleted from the table. Assuming the id field is your primary key, this query matches at most one record. If no records with an id of 15 are found, no changes are made. Note the use of the *commit* statement to make the changes permanent.

The example syntax uses the standard Python driver method of writing the SQL statement yourself. Field values are denoted with a percent sign in the first parameter, the SQL. The second parameter is a tuple (or list) of values to be inserted into the SQL statement upon execution. This is the preferred way of constructing SQL statements manually because it allows the driver to handle special characters (like commas or quotes) automatically. Note that the second %s in the string does not require single quotes, even though the *nm* variable is a string.

Readers may note the seemingly extra comma in the parameter list: `(id, )`. This comma is required to tell Picalo that the parameter is a tuple (e.g. read-only list). If you leave the comma off, Picalo will ignore the parentheses and, depending upon the database driver you are using, may fail.

5.11 Delete All Records From a Database

Picalo GUI Recipe:

1. Not available.

Script Recipe:

```
# delete the records, but keep the (now empty) table
conn = Database.OdbcConnection('mydb', username='me', password='ps')
conn.execute("DELETE FROM mytable")
conn.commit()
conn.close()

# drop the entire table from the database
conn = Database.OdbcConnection('mydb', username='me', password='ps')
conn.execute("DROP mytable")
conn.commit()
conn.close()
```

Discussion:

Because the first example contains no WHERE clause, all records in the database match the query and are deleted. Note the use of the *commit* statement to make the changes permanent.

5.12 Access a Database Directly (bypassing Picalo)

Picalo GUI Recipe:

1. Not available.

Script Recipe:

```
# import the driver and connect to the database
import psycopg2
conn = psycopg2.connect('mydb', username='me', password='ps', host="10.5.2.1")

# create a cursor and run the query
cursor = conn.cursor()
cursor.execute("SELECT id, name FROM mytable")

# iterate through the results
for rec in cursor:
  print 'id is:', rec[0]
  print 'name is:', rec[1]

# close up shop
cursor.close()
conn.close()
```

Discussion:

There may be times that you want to bypass Picalo entirely and use the database driver directly. Perhaps Picalo adds too much overhead to your query. Or perhaps you need to run a driver command directly. Even though you might enter the code in the Picalo shell or run a script from the Picalo run command, your code always runs directly in Python. Picalo commands are only used when you explicitly call them.

The above example shows how to connect to a PostgreSQL database directly (the command varies by database driver), query a result set, and print each record. Note that since you are bypassing Picalo, you cannot access field values by column name, cannot use most Picalo functions, and can only access the records sequentially.

5.13 Create Unique Numbers

Picalo GUI Recipe:

1. Not available.

Script Recipe:

```
# import the GUID library
from picalo.lib import GUID

# create a few guids and append to a list
myguids = []
for i in range(len(10)):
    g = GUID.generate()
    myguids.append(g)

# print information about the GUIDs
for g in myguids:
    print 'IP:', GUID.extract_ip(g)
    print 'Time', GUID.extract_time(g)
    print 'Counter:', GUID.extract_counter(g)
```

Discussion:

Picalo contains a GUID generator library. GUIDs are globally-unique IDs that are unique in space and time. Each GUID encodes the IP address of the computer where it was created, the millisecond (since 1970), and a sequence number (in case two GUIDs are created in the same millisecond on the same computer).

GUIDs are generally useful as database keys, network packet identifiers, web cookies, and other places where unique numbers are needed. When used as database keys, they circumvent the need to keep a running counter, run a "max" query on the primary key field, or use a non-portable autoincrementing field. Using GUIDs for database keys also allows you to combine database tables together (for example, from two databases) without getting key clashes and without having to rewrite primary and foreign keys.

GUIDs are not perfectly unique. If two processes on the same machine create GUIDs at the same time, you might get a clash. If two machines have the same IP address (i.e. private IP addresses might do this), they might create clashing GUIDs. But if you control for these factors, GUIDs give you a quick, portable, way to generate unique numbers.

Chapter 6

Scripting

Scripting is your highway to automation and macros in Picalo. Picalo is based upon a very powerful, popular language: Python. By using a standard language, Picalo inherits a significant amount of functionality and power, including a large module library, true object-orientation, and web and filesystem abilities.

I think you'll find the Python language well-written and easy to learn. It is widely regarded as one of the 'prettier' languages on the market, and it has served as an excellent foundation for Picalo.

6.1 Run a Command in the Shell

Picalo GUI Recipe:

1. Ensure the Shell tab is visible in the bottom right area of the Picalo window. If you can see the tab but can't see the Shell window, you might need to drag the window splitter up to reveal the window.
2. Click the mouse to the right of the bottom-most `>>>` indicator. You can only type commands on the bottom-most indicator line.
3. Type your command and press Enter.

Discussion:

The Shell is one of the most powerful features of Picalo. Using the Shell, you can enter Picalo commands directly. This is useful when you are testing out functions that you'll use in a larger script later and when you need to enter one or two additional commands after a script finishes.

Memory and variables are shared by all parts of Picalo: the GUI, the Shell, and scripts. In you create tables in your scripts, you'll see them listed in the left-side object browser after the script finishes. If you modify a table using a Shell command, you'll immediately see it reflected in the viewer area.

The Picalo GUI, including all dialog boxes, is actually just a front end to the Shell. As you use the dialogs of Picalo, you'll see the corresponding commands run automatically in the Shell. This is transparent and shown to you so you can learn the commands directly. The goal of the GUI is to help you start scripting (where you can really use Picalo's power). The Shell is an intermediate step in this goal.

6.2 View the History

Picalo GUI Recipe:

1. Click the History tab in the bottom right area of the Picalo window. If you can see the tab but can't see the actual window, you might need to drag the window splitter up to reveal the window.

Discussion:

The history keeps track of every command (including GUI dialogs) you run in Picalo. This allows you to repeat your commands at a later date, convert your commands to a script, and document your work. If your work with Picalo involves legal action, keeping a log of your work is vital.

6.3 Save the History

Picalo GUI Recipe:

1. Select *Edit* — *Save History...*
2. Enter a filename and click Save.

Discussion:

The history keeps track of every command (including GUI dialogs) you run in Picalo. This allows you to repeat your commands at a later date, convert your commands to a script, and document your work. If your work with Picalo involves legal action, keeping a log of your work is vital.

6.4 Start a New Script

Picalo GUI Recipe:

1. Select *Script* — *New Script* or click the little blue folder in the left end of the toolbar.

Discussion:

Picalo scripts are by far the most powerful way to use Picalo. Scripts allow you to combine the Picalo data analysis functions with the excellent Python language. They are similar to (but more powerful than) macros on other platforms.

To begin scripting, watch the Shell as you use the Picalo GUI. As you watch the Picalo functions being used, you'll get an introduction to the commands that can be used in scripts.

The Picalo editor is only a basic text editor. It does not have the powerful features you might find in dedicated editors like TextWrangler (Mac), Notepad++ (Windows), or many other powerful editors on the market. Picalo supports your use of these editors. If you load a script into *both* Picalo and your favorite editor, Picalo will automatically detect when you make changes to the file. When it detects that you've made changes in a separate application, it will prompt you to reload the file from disk.

6.5 Run a Script

Picalo GUI Recipe:

1. Load or create a script using the *Script* menu. You should see your script in the viewer area.
2. Select *Edit — Run Script* from the menu or click the blue triangle icon on the toolbar.

Discussion:

Scripts should normally be run directly in Picalo. This allows you to use Picalo's viewer area, existing variables (like database connections created before the script run), and Picalo's editor to create and run your script. When the script is finished, any variables you create will be available for further analysis and inspection.

This shared-memory model is by design. Most analysts want their data available after a script run. The drawback of this model is your script may overwrite existing variables in memory. If you are worried about overwriting existing variables, run your script in a new Picalo instance (Recipe 6.6).

6.6 Run a Script in New Picalo

Picalo GUI Recipe:

1. Load or create a script using the *Script* menu. You should see your script in the viewer area.
2. Select *Edit* — *Run Script In New Picalo* from the menu or click the blue triangle icon on the toolbar.

Discussion:

Because Picalo shares memory between scripts, the GUI, and the Shell, variables might step on one another. While this shared-memory model is normally desirable, Picalo allows you to run your scripts within a fresh instance of the program with a single command. This ensures that your script starts with nothing previously existing. It also allows you to terminate the new Picalo (if needed) without affecting your existing Picalo run.

6.7 Run a Script Outside of Picalo

Picalo GUI Recipe:

1. Not available (obviously if you want to do this outside of Picalo! :)

Discussion:

There may be times that you want to remove the Picalo GUI entirely. This allows you to use Picalo functions like Database, Grouping, Simple, etc. within other programs, as part of a cron or scheduler job, or directly from the command prompt/terminal. There are many organizations that I've worked with that are using Picalo in this manner on their servers.

In fact, Picalo was originally meant to run as embedded software and had no GUI. Since that time, I have made sure that Picalo can continue to be run this way. It hasn't always been easy to ensure the GUI is not required, but it has been done.

To run Picalo without the GUI, complete the following steps:

- Install Python from `http://www.python.org/`. Windows users should install ActivePython, an extension to Python that integrates with Windows better than standard Python. *nix and Mac users probably already have Python installed (go to the terminal and type "python" to find out).
- Download the *source* distribution of Picalo from `http://www.picalo.org`. Unzip the tar.gz file to an empty directory somewhere.
- Open a command prompt (Windows) or a terminal window (Mac/*nix). Navigate to the directory where you unzipped Picalo. This directory should have a `setup.py` file in it.
- Type `python setup.py install`. Picalo will install into your Python libraries.
- To test your installation, start Python (type "python" at the command prompt) and type `from picalo import *`.

- Now that Picalo is installed, simply include the line `from picalo import *` at the top of your scripts. You'll be able to run them just like any other Python script.

6.8 Cancel a Running Script

Picalo GUI Recipe:

1. Not available.

Script Recipe:

```
# go through a table one record at a time
for i, rec in enumerate(mytable):
    # show the user progress
    # THIS is what allows the user to cancel the script
    perc = float(i) / float(len(mytable))
    show_progress(perc, 'Processing the table')

    # do your analysis here
```

Discussion:

Picalo utilizes a shared-memory model for running scripts, which means that scripts run within the Picalo environment. When scripts are running, all of Picalo's attention is given to the running script. The only part of the Picalo window that will update is the output area (i.e. when you use the *print* statement to print results. The rest of Picalo will appear frozen.

The alternative approach (the one used by most development environments) is to run scripts in a separate process. This would allow the Picalo GUI to remain responsive and would allow you to simply kill the process at any time by clicking a 'Stop' button. However, running in a separate process would not allow memory sharing to occur. One of the strengths of Picalo is the connection between scripts and the GUI. Anything you do in scripts is reflected in the left-side object browser and in the table views. When scripts are finished, you can continue to analyze your data because the variables and tables remain in memory. If you truly wish to run a script in a separate, stoppable process, select *Run Script In New Picalo* to start a new Picalo instance and memory space. You can then terminate the new Picalo at any time through your regular operating system methods.

The preference at most times is for Picalo and scripts to share memory. This means that Picalo cannot simply kill scripts at will – doing so would leave system resources in a dirty state and would make Picalo unstable. The undesirable effects of terminating threads directly is a well-documented computer science phenomenon. This is why Picalo doesn't include a 'Stop' button on its interface.

Now that I've (hopefully) convinced you that there are good reasons to limit the direct termination of scripts, let me explain the workaround. The 'right' way to stop a script is to raise an exception so your scripts exits on their own. The easiest way to do this is to show a progress bar to the user. The standard Picalo progress dialog has a 'Cancel' button.

Every time you call *show_progress*, Picalo will check to see if the user has hit the cancel button. If the user has canceled the operation, Picalo will raise an exception and your script will exit immediately.

For this to work, though, you need to call *show_progress* often in your script. The method is efficient and will not significantly affect your script speed. If you simply show a progress dialog at the beginning of your script and never update it, Picalo won't have a chance to check the cancel button.

The bottom line is to call *show_progress* before each part of your analysis, within your for and while loops, and everywhere else it makes sense to show the user updated progress. The user will thank you anyway because users always like to see the progress of their analyses.

6.9 Use a Standard Python Module

Picalo GUI Recipe:

1. Not available.

Script Recipe:

```
1 import random
2
3 for i in range(100):
4     print random.randint(0, 1000)
```

Discussion:

The functions of Python are organized according to modules. Rather than loading all available functions in to memory each time your run a script, Python only loads a few standard functions. Any additional functions you want to use must be *imported* into your script. This separation of functions into different modules provides for organization and efficient memory usage.

The standard Python libraries are described in the Python documentation at `http://www.python.org/`. Readers are encouraged to investigate these libraries to learn about the many functions available. Thousands more additional libraries can be found on various web sites.

When you create a new script, Picalo inserts a few *import* statements at the top of your file. The first statement, `from picalo import *`, imports the primary Picalo libraries for use in your script. The second statement imports commonly-used Python functions, such as the following:

- sys: Access to system resources and information.
- re: Regular expression libraries for powerful, advanced text searching.
- datetime: Access to Python's built-in date and time library. Picalo uses this library for the built-in DateTime and Date types.
- os: Functions related to files, directories, and system processes.

- os.path: Functions for parsing file and directory names.
- urllib: Allows reading of web pages; useful for ethical scraping techniques.

6.10 Use a Nonstandard Python Module

Picalo GUI Recipe:

1. Not available.

Script Recipe:

```
# See discussion below for installation, then import normally:

import mymodule
```

Discussion:

Picalo is released as a standalone program (exe for Windows and app for Mac). The python interpreter bundled with the application is frozen when compiled, and it can't be changed directly. Therefore, only the modules included in the frozen build (at compile time) are normally available.

For those advanced users who want to use additional modules in their code, there is a workaround available. Additional modules might be the Python Imaging Library, an additional database driver, the python interface to R, or any number of modules available on the Internet. To install an additional library not normally included with Picalo, use the following steps:

1. Install the regular version of Python (i.e. from `http://www.python.org` or ActivePython on your computer. Advanced users will probably already have a standard Python interpreter installed. Linux and Mac users also have Python installed by default.

2. Install your library into the standard Python interpreter (not into Picalo). This is normally done with `python setup.py install` or some variation thereof.

3. Start your standard Python interpreter and ensure that the new module works as expected.

4. Append the `site-packages` directory from your standard Python installation to `sys.path` when in Picalo. For example, suppose you install a module called *mymodule* into your regular Python installation and want to use it in Picalo. Once it is installed correctly, start Picalo and type the following:

```
# append to the system module path
sys.path.append('c:/python/site-packages')

# import normally
import mymodule
```

5. Use your module within Picalo!

The `site-packages` directory location varies depending upon where your Python interpreter is installed. For example, the standard Mac Leopard location is /Library/Python/2.5/site-packages. An easy way to find out where your new module is installed is to check the __file__ attribute of your module. For example, in your regular Python interpreter (not in Picalo), type the following:

```
import mymodule
print mymodule.__file__
```

This will tell you what directory needs to be included in sys.path to make things work.

For now, you need to add the module path to sys.path every time you start Picalo.

6.11 Create Function Libraries

Picalo GUI Recipe:

1. Not available.

Script Recipe:

mathlib.py

```
1 def divide(numerator, denominator):
2     '''Returns the quotient and remainder of division'''
3     quotient = int(numerator) / int(denominator)
4     remainder = int(numerator) % int(denominator)
5     return quotient, remainder
```

myscript.py

```
1 import mathlib
2
3 q, r = mathlib.divide(5, 2)
4 print 'Quotient is', q
5 print 'Remainder is', r
```

Discussion:

Picalo allows you to centralize common functions into library scripts, called *modules*. Modules are simply regular scripts that have their code expressed as functions and that are imported into other scripts. In the example above, the *divide* function is declared to have two parameters. At the end of the function, it returns two values. A module can have as many functions as you wish to place in it. Modules can also contain classes, global variables, and just about anything else a regular script contains. The example above is named "mathlib.py".

The functions, variables, and classes in a module are imported into another script by the *import* command. Functions from imported modules are called using the modulename.functionname format as shown in the example above. Notice how the "myscript.py" script imports the module name without the *.py* extension. You can think of the import command as a virtual cut and paste operation. At the point of the import command in myscript,

all of the code from the mathlib.py module are essentially included in the file, as if they had been in myscript.

Normally, a script must be in the same directory as an importing script. Python contains a search path for module files, and the top-level script is at the first of this search path. For advanced users, you can actually modify the search path using *sys.path*, but modifying the search path could make Picalo unstable (for example, if you remove the path to Picalo's modules). It is usually safest to simply place the two scripts (the module and the importing script) in the same directory.

6.12 Show Script Progress to the User

Picalo GUI Recipe:

1. Not available.

Script Recipe:

```
# go through a table one record at a time
for i, rec in enumerate(mytable):
    # show the user progress
    perc = float(i) / float(len(mytable))
    show_progress(perc, 'Processing the table')

    # do some analysis with the table here
    rec.Name = rec.FirstName + ' ' + rec.LastName
```

Discussion:

Picalo contains a standard progress dialog that your scripts can use. The *show_progress* command shows the dialog with a custom message and a percent done bar. The percent done should always be given as a floating point number between 0 and 1. If the percent done is not in this range (i.e. greater than or equal to 1), the progress dialog is removed.

The progress dialog only shows when you are running the GUI version of Picalo (as most users do). If you are running a script directly in your terminal (i.e. from Bash or the Command Prompt), the command will be ignored by Picalo. In other words, if you include the *show_progress* command, your code will work whether or not the progress dialog can be shown or not.

Showing a progress bar is important because it allows the user to cancel your script at any time. It also provides feedback so the user knows the script is running successfully and is not frozen.

6.13 Turn Off Picalo Progress Indicators

Picalo GUI Recipe:

1. Not possible

Script Recipe:

```
1 use_progress_indicators(False)
```

Discussion:

There may be times when you want Picalo to go silent. Normally, Picalo uses progress indicators, both in GUI and text mode, to let you know how an analysis is going. Turn off indicators with the *use_progress_indicators* function. Turn them back on with a True parameter.

6.14 Show a File Selector to the User

Picalo GUI Recipe:

1. Not available.

Script Recipe:

```
1 import wx, os
2
3 filename = wx.FileSelector("Select a File", flags=wx.OPEN | wx.FILE_MUST_EXIST)
4 if filename:
5   # parse the directory and filename
6   dirname, fname = os.path.split(filename)
7   # parse the filename part from the extension part
8   fn, ext = os.path.splitext(fname)
9
10  # do something with the filename, such as
11  for line in open(filename):
12    line = line.strip()
13    print line
```

Discussion:

The wx toolkit that Picalo is built with is available to your scripts. It provides ways to build full dialogs, graphical user interfaces, and dialogs. The documentation for wx is located at `http://www.wxpython.org/`.

The above example shows how to get a filename from the user. The wx documentation for this function explains how to set a default filename, use a filename mask for the dialog, show a save or open dialog, and set the default directory in the dialog.

Chapter 7

Text Processing

7.1 Read an Entire Text File

Picalo GUI Recipe:

1. Not available.

Script Recipe:

```
1 f = open('c:/temp/myfile.txt')
2 data = f.read()
3 f.close()
```

Discussion:

Picalo can read text files in several ways using the *open* command. The example shows how to read an entire file into a string variable. The data variable in the example holds the entire bytes of the file, including new line characters. The *open* command returns a file object, which is a pointer to the file bytes on disk.

A more effective way of reading text files is usually line by line, which is shown in the next recipe.

7.2 Read a Text File Line By Line

Picalo GUI Recipe:

1. Not available.

Script Recipe:

```
# read a file efficiently
f = open('c:/temp/myfile.txt')
for line in f:
    line = line.rstrip()
    print line
f.close()

# read a file into a list of file lines
f = open('c:/temp/myfile.txt')
lines = f.readlines()
f.close()
```

Discussion:

Picalo can read text files in several ways using the *open* command. The first example shows the most efficient way to read files – line by line. Since only a single line is read at a time, the operation takes very little memory, even for large files. The *open* command returns a file object, which is a pointer to the file bytes on disk. Since file objects can act like regular Python lists, the code is able to iterate across the file lines using the familiar *for* loop.

The *rstrip()* command is issued immediately after reading a new line to strip off any trailing white space. When Picalo reads files lines, it does not remove the new line character, which is character 10 on Unix/Mac and characters 13 and 10 on Windows. Stripping the right side of the line removes any extraneous white space, which includes new line characters, tabs, and spaces.

The second example shows a less efficient but easier way to read files. If you have a relatively small file (that can be held in memory all at once), the *readlines* function will turn the file lines into a real Python list. You can then use all the familiar methods of lists, access arbitrary locations like lines[2] for the third line, sort the lines, etc.

7.3 Import Email Into Picalo

Picalo GUI Recipe:

1. Select File — Import — Email...
2. Follow the wizard.

Script Recipe:

```
1 import email
2
3 msg = email.message_from_file(open('an_email.txt'))
4 print msg['To']
5 print msg['Subject']
6 for part in msg.walk():
7     if part.get_content_maintype() == 'multipart':
8       continue
9     print '=========================='
10    print part
```

Discussion:

Picalo contains a wizard to import both mbox-formatted and Maildir-formatted email messages. For Outlook, Notes, or GroupWise email, search the web for a converter program. For example, many converters exist to convert Outlook's .pst file format to mbox format. Once you get the email in the right format, Picalo's import wizard will create a table for the entire mail folder.

If you prefer to use code, Python's standard *email* module can parse many different types of email. In the example script, the parts of a Maildir-style of email is walked. Maildir style is a type of email storage on Unix servers that keeps each email in a separate file in the user's home directory. It is one of the most common email storage formats in the world. Other email systems, such as MS Exchange, keep emails in a relational database (these email systems are best accessed by administrator programs).

With a few modifications, the above code can also read mbox-style files. The file would first need to be split on the new line + 'From' keyword. Use

the string *split* method to split the Mbox file into a list of individual emails, then run the above algorithm on each item in the list.

The email module creates a message object, termed 'msg' in the example. Message headers are accessed using the dictionary interface, such as *msg['To']* to get the recipient information. The parts of the email, such as the main body, attachments, an html-version of the email, and a plain-text version of the email, are accessed by walking the parts.

Using the above recipe, you could easily walk a large set of emails and import them into a structured Picalo table or into a database. If you need to go through many files in a directory (i.e. Maildir style), use the *os.listdir* function to get all the files in the directory.

Finally, since Picalo is an open source application, advanced users might want to look at the import wizard script directly. It is in the picalo/gui/wizards directory of the source distribution.

7.4 Extract Data From Nonstandard Text Files

Picalo GUI Recipe:

1. Use the data import wizard to import the data. Nonstandardized text files require scripts, as shown in the example below.

Script Recipe:

```
# import the regular expression library
from picalo import *
import re

# create a table to hold the data
data = Table([
  ( 'fname', str ),
  ( 'lname', str ),
  ( 'mname', str ),
  ( 'suffix', str ),
])

for line in open('names.txt'):
    # first remove extraneous white space and newline characters
    name = line.strip()

    # check for last name alone
    match = re.search('^(\w+)$', name)
    if match:
        newrec = data.append()
        newrec.lname = match.group(1)
        continue

    # check for first last
    match = re.search('^(\w+)\s+(\w+)$', name)
    if match:
        newrec = data.append()
        newrec.fname = match.group(1)
        newrec.lname = match.group(2)
        continue

    # check for first mi last
    match = re.search('^(\w+)\s+(\w)\s+(\w+)$', name)
    if match:
        newrec = data.append()
        newrec.fname = match.group(1)
        newrec.mname = match.group(2)
        newrec.lname = match.group(3)
        continue

    # check for first last suffix
```

```
42      match = re.search('^(\w+)\s+(\w+)\s+(\w+)$', name)
43      if match:
44          newrec = data.append()
45          newrec.fname = match.group(1)
46          newrec.lname = match.group(2)
47          newrec.suffix = match.group(3)
48          continue
49
50  # view the table
51  data.view()
```

Discussion:

There may be times that you cannot use Picalo's standard data import routines (tsv, csv, fixed, etc.) because your data does not match any common standard. These type of data files are not difficult to read; just use Python's powerful regular expression library to extract the fields. Regular expressions are very powerful and are not trivial to utilize. You should read the standard Python documentation for regular expressions for more information on this powerful text analysis tool.

The first step is to read the data file line by line (assuming each line represents a record). On each record, try to match the line using a regular expression pattern. When you get a match, extract the fields using groups. Groups are defined in your regular expression pattern with parenthases.

The example pattern above, we assume that an input file has a number of employee names. We want to split the full names into the first, middle, last, and suffix. The file is difficult to read with standard methods because of the following:

- Some lines contain only last names, some contain first+last, some contain first+middle initial+last, and some contain first+last+suffix. Because of this lack of standardization, the file is difficult to read.
- Each line contains a number of spaces between the name parts. Some contain only one space and others contain many. There is no way to predict how many spaces are between name parts.

The patterns in the script use the following matching characters:

- \w: This character represents a single letter or number.

- \s: This character represents a single white space character (e.g. a space)
- +: The plus indicates that the previous character can be repeated one or more times. The plus is always used in combination with another character, such as \s, which matches one or more spaces.
- ^ and $: These characters match the start and end of each line. If we do not specifically mark the beginning and end of the line, the pattern will match parts of lines, which we do not want to allow in this case. We want to match all characters on the line, especially since we are stripping any extra white space at the beginning of each loop iteration.
- **parenthases**: These mark groups, which is how the script pulls data from each line. Notice that the number of matched parentheses always matches the group statements below each pattern. The first pair of parentheses is group(1), the second is group(2), and so forth.

The script will iterate through each line of the file. On each line, it tries first to match the last name alone. If successful, it records the name into the data table and continues to the next line. If unsuccessful, it continues to the next match. Since the matches are given in order of increasing specificity, the script is able to extract the data.

Notice that the only difference between the third and fourth patterns is a single plus sign. Both the first+mi+last and first+last+suffix lines have three words separated by spaces. Since the middle initial is only one character (notice the lack of the plus on the pattern), we can differentiate between the two lines.

This recipe is only an introduction to what regular expressions can do. Regular expressions were invented through linguistics research many decades ago – before the advent of computers. They are by far the most powerful text searching and matching language I've found. They are probably your best bet in matching and searching any kind of text.

Regular expressions are the most common matching method used in computer programming; support for standard regular expressions can be found in most modern languages. If you do a lot of text matching, I suggest you purchase a book on regular expression syntax.

Chapter 8

Other Useful Tasks

This chapter contains recipes for useful tasks that do not fit into other chapters.

8.1 Generate Random Numbers

Picalo GUI Recipe:

1. Double-click a table in the left-side project browser to view the table.
2. Select *File — Table Properties* .
3. Add a new column and set its type to 'Calculated Field'.
4. Enter the following calculation: random.randint(100, 500).

Script Recipe:

```
1 a = random.randint(100, 500)
2 b = random.uniform(100, 500)
3 table.append_calculated("random.randint(100, 500)")
4 table.append_calculated_static("random.randint(100, 500)")
```

Discussion:

Random number generation can be useful when sampling tables, running dynamic analyses, using genetic algorithms, or in many other instances. Python's *random* library can generate many different types of numbers, including integers and floating-point numbers. In addition, it can match several distributions like uniform, betavariate, expovariate, gammavariate, guass, and many others.

The GUI recipe adds a calculated column to a table. The expression in the new column uses the random library to generate a random integer between 100 and 500. An alternative expression might have been `random.uniform(100, 500)` to generate a decimal number in the same range.

The GUI recipe above has a catch: since calculated fields added with Picalo's table properties dialog are dynamic, the random numbers will be regenerated *each time you view the table.* If you need the numbers to be stable and unchanging, use *append_calculated_static* from the shell rather than the table properties dialog.

The shell recipe shows several different examples of generating random numbers. The first two (randint and uniform) create a random integer and floating-point number, respectively. The third example appends a dynamic column of random numbers (read the catch in the previous paragraph). The fourth and final line adds an unchanging column of random numbers. Once generated, these numbers remain the same throughout the life of the table.

8.2 Randomize Table Records

Picalo GUI Recipe:

1. Not available.

Script Recipe:

```
1 random.shuffle(table)
```

Discussion:

Suppose you have a table and want to randomize the records. In other words, you want the records shuffled into a new, random order. One application of this technique is to get a random sample of records in a table. Just shuffle the table, then take the top n records for your sample.

The technique uses the *shuffle* function in Python's random library. Once you have shuffled a table, there is no way to return to the previous order (unless you can sort the table to its previous order using a primary key column).

8.3 Choose a Random Table Record

Picalo GUI Recipe:

1. Not available.

Script Recipe:

```
# this method simply selects a random record
rec = random.choice(table)

# this method copies a single record to a new table,
# then selects the record
randnum = random.randint(0, len(table))
rec = table[randnum: randnum+1][0]
```

Discussion:

The first method uses the *choice* method of the random library, which selects a random element from a list (since a table is a list of record objects, this works). Be aware that the record *is still part of the table*; any changes you make to the record will be reflected in the source table.

The second method selects a copy of a record rather than the source record. It does this by using [a: b] notation to first generate a selected copy of the source table. It then selects the record from this new table. This method, while more complex, allows you to make changes to the record without affecting the source data.

8.4 Scrape a Web Page for Data

Picalo GUI Recipe:

1. Not available.

Script Recipe:

```
import re, urllib

# suppose the HTML of the web page we're looking for is:
'''
<html><body>
Welcome to my weather station site.  I run a weather station for my area,
and it sample the weather conditions every few minutes.  Here's the data:
<ol>
  <li>Current Temperature: 14 degrees</li>
  <li>Current Wind: 5 kph NW
</ol>
</body></html>
'''

# first step is to download the text from the web site
f = urlopen('http://www.myweatherserver.com/current.html')
text = f.readlines()
f.close()
# the text variable now contains the HTML of the web page!
# we assume it's the string at the top of this script (for demo purposes)

# there are many ways to do this, but we'll just go
# through the file one line at a time
for line in text:
  match = re.search('Temperature: (\d+) degrees')
  if match:
    print 'Current weather is', match.group(1)
```

Discussion:

Web page scraping is as old as the World Wide Web. It is the method by which search engines spider the web to populate their searches, the way spammers get lists of email addresses, and the method that clever analysts get needed data.

The problem is the web presents human-readable data; it's unorganized, free text. Suppose a web site publishes information that you need for an

analysis. It might be weather data, law enforcement information, or any other type of data. You should first try to find the data in a structured format – some web sites give a link for their data in XML, CSV, TSV, or other format. If no structured source is available, you have two choices left: either go through the pages by hand, one by one, or write a short program to do the walking for you.

Web scraping is an inherently fragile process. The source web site publisher is free to change the data anytime he or she wishes (breaking your scraper). Most web sites track the number of hits on their site, and when your robot quickly hits thousands of their web pages, they might kick you off their web server. Web scraping should be used as a last resort.

Before talking about how to do this, I should discuss the ethics of scraping web pages. On the one hand, web scraping makes the web work – we wouldn't have search engines without it. On the other hand, web scraping has been the source of a significant amount of spam. In effect, you are going to write a program that will mimic your web browsing – going to sites and searching for information very quickly. Instead of your going to each web page, your program will do it, analyze the HTML text, and grab the information you're looking for. Web scraping can bring down web servers because of repeated hits, violate privacy, and cause general havoc if used incorrectly.

Most web server provide a file, ROBOTS.TXT, in their web root. Simply go to `http://www.server.com/ROBOTS.TXT` (case sensitive) where *server* is the name of the server you want to scrape. The file will tell you which parts of the web site can be scraped by robots and which parts cannot. If no file exists, the site is generally seen as fair game. Note that no one is enforcing this file – web ruffians certainly don't abide by them. In fact, Google (which makes its living by scraping everyone else) allows almost no scraping of their own site in their ROBOTS.TXT file.

Assuming you feel it is ethical for you to scrape a web site for data, Python's *urllib* and *re* libraries make it easy and fast to do so. In this example, I'm only scraping a single web page, but it would be easy to expand the example to scrape thousands of pages using a simple for loop.

The first step is to download the page text into a variable. Use *urllib.urlopen* to open the web connection, then use *readlines* to read each line into a list. Close up the connection. The second step is to use regular expressions to pattern match the data you are looking for. See the *re* module for more information.

Note that the regular expression uses parentheses to group the current temperature. The text surrounding this group (the keywords 'Temperature:' and 'degrees') help the regular expression engine know what numbers to find. If we find a match on the line, calling `group(1)` gives us the first match.