Larch Documentation

Release 0.9.8

Matthew Newville

March 05, 2012

CONTENTS

Larch is a scientific data processing language that is designed to be

- easy to use for novices.
- complete enough for intermediate to advanced data processing.
- data-centric, so that arrays of scientific data are easy to use and organize.
- easily extensible with python.

Larch is targeted at tools and algorithms for analyzing x-ray spectroscopic and scattering data, especially the sets of data collected at modern synchrotrons. It has several related target applications, all meant to be better connected through a common *macro language* for dealing with scientific data sets.

Many data collection, visualization, and analysis programs have an ad-hoc macro languages built into them that allow some amount of customization, automation, scripting, and extension of the fundamental operations supported by the programs. These macro languages are rarely used in more than one program, making communication and sharing data between programs very hard.

Larch is an attempt to make a macro language that can be used for many such applications, so that the algorithms and techniques for visualization and analysis can be better shared between different programs and fields. In this respect, Larch is meant to be the foundation or framework upon which data collection, visualization, and analysis programs can be written. By having a common, extensible macro language and analysis environment, the hope is that it will be easier to make data collection, visualization, and analysis programs interact.

Larch is written in Python, has syntax that is quite closely related to Python, and makes use of many great efforts in Python, especially for scientific computing. These include numpy, scipy, and matplotlib.

The initial target application areas for Larch are

- XAFS analysis, becoming version 2 of Ifeffit.
- tools for micro-XRF mapping visualization and analysis.
- quantitative XRF analysis.
- X-ray standing waves and surface scattering analysis.

Contents:

CHAPTER

ONE

DOWNLOADING AND INSTALLATION

1.1 Prerequisites

Larch requires Python version 2.6 or higher. Support for Python 3.X is partial, in that the core of Larch does work, and numpy, and scipy, and matplotlib have all been ported to Python 3.X. But the testing for Python 3.X has been minimal, and the graphical interfaces, based on wxWidgets, has not yet been ported to Python 3.X.

Numpy, matplotlib, and wxPython are all required for Larch, and scipy is strongly encouraged (and some functionality depends on it). These are simply installed as standard packages on almost all platforms.

All development is being done through the larch github repository. To get a read-only copy of the atest version, use:

```
git clone http://github.com/xraypy/xraylarch.git
```

1.2 Installation

Installation from source on any platform is:

```
python setup.py install
```

We'll build and distribute Windows binaries and use the Python Package Index soon....

1.3 License

This code and all material associated with it are distributed under the BSD License:

```
Copyright, Licensing, and Re-distribution
-----------------------------------------

Unless otherwise stated, all files included here are copyrighted and
distributed under the following license:

  Copyright (c) 2010-2011 Matthew Newville, The University of Chicago

  Redistribution and use in source and binary forms, with or without
  modification, are permitted provided that the following conditions are met:

  * Redistributions of source code must retain the above copyright notice,
    this list of conditions and the following disclaimer.
  * Redistributions in binary form must reproduce the above copyright
```

CHAPTER

TWO

LARCH TUTORIAL

This chapter describes the Larch language and provides an introduction into processing data using Larch. An important goal of Larch is to make writing and modifying data analysis as simple as possible. The tutorial here tries to make few assumptions about your experience with scientific programming. On the other hand, Larch is a language for processing of scientific data, the expected audience is expected to have a technical background, familiarity with using programs for scientific data analysis. In addition, some understanding of the concepts of how scientific data is stored on computers and of the basics of programming.

The Larch language is implemented in and heavily based on Python. Knowledge of Python will greatly simplify learning Larch, and vice versa. This shared syntax is intentional, so that as you learn Larch, you will also be learning Python, which can be used to extend Larch. Alternatively, knowledge of Python will make Larch easy to learn. For further details on Python, including tutorials, see the Python documentation at http://python.org/

Contents:

2.1 Tutorial: Getting Started

This tutorial expects that you already have Larch installed and can run either the program larch, basic Larch interpreter, or larch_gui, the enhanced GUI interpreter:

```
C:> larch
  Larch 0.9.7  M. Newville, T. Trainor (2011)
  using python 2.6.5, numpy 1.5.1
larch>
```

For Windows and Mac OS X users, executable applications will be available.

2.1.1 Larch as a Basic Calculator

To start with, Larch can be used as a simple calculator:

```
larch> 1 + 2
3
larch> sqrt(4.e5)
632.45553203367592
larch> sin(pi/3)
0.8660254037844386
```

You can create your own variables holding values, by assigning names to values, and then use these in calculations:

```
larch> hc = 12398.419
larch> d = 3.13556
larch> energy = (hc/(2*d)) / sin(10.0*pi/180)
larch> print energy
11385.470119348252
larch> angle = asin(hc/(10000*2*d))*180/pi
larch> print angle
11.402879992850263
```

Note that parentheses are used to group multiplication and division, and also to hold the arguments to functions like `sin()`.

Variable names must start with a letter or underscore ('_'), followed by any number of letters, underscores, or numbers. You may notice that a dot ('.') may appear to be in many variable names. This indicates an *attribute* of a variable – we'll get to this in a later section.

If you're familiar with other programming langauges, an important point for Larch (owing to its Python origins) is that variables are created *dynamically*, they are not pre-defined to have some particular data type. In fact, the a variable name (say, 'angle' above) can hold any type of data, and its type can be changed easily:

```
larch> angle = 'now I am a string'
```

Although the types of values for a variable can be changed dynamically, values in Larch (Python) do have a definite and clear type, and conversion between types is rigidly defined – you can add an integer and a real number to give a real number, but you cannot add a string and a real number. In fact, writing:

```
larch> angle = asin(hc/(10000*2*d))*180/pi
```

is usually described as "create a variable 'angle' and set its value to the result calculated (11.4...)". For those used to working in C or Fortran, in which variables are static and must be of a pre-declared type, this description is a bit misleading. A better way to think about it is that the calculation on the right-hand-side of the equal sign results in a value (11.4...) and we've assigned the name 'angle' to hold this value. The distinction may seem subtle, but it can have some profound results, as we'll see in the following section when discussing lists and other dynamic values.

2.2 Tutorial: Basic and Complex Data Types

This section of the Larch tutorial describes the types of data that Larch has, and how to use them.

2.2.1 Basic Data Types

As with most programming languages, Larch has several built-in data types to express different kinds of data. These include the usual integers, floating point numbers, and strings common to many programming languages. A variable name can hold any of these types (or any of the other more complex types we'll get to later), and does not need to be declared beforehand or to change its value or type. Some examples:

```
larch> a = 2
larch> b = 2.50
```

The normal '+', '-', '*', and '/' operations work on numerical values for addition, subtraction, multiplication, and division. Exponentiation is signified by '**', and modulus by '%'. Larch uses the '/' symbol for division or 'true division', giving a floating point value if needed, even if the numerator and denominator are integers, and '//' for integer or 'floor' division. Thus:

```
larch> 3 + a
5
```

```
larch> b*2
5.0
larch> 3/2
1.5
larch> 3//2
1
larch> 7 % 3
1
```

Several other operators are supported for bit manipulation.

Literal strings are created with either matched closing single or double quotes:

```
larch> s = 'a string'
larch> s2 = "a different string"
```

A string can include a 'n' character (for newline) or 't' (for tab) and several other control characters as in many languages. For strings that may span over more than 1 line, a special "triple quoted" syntax is supported, so that:

```
larch> long_string = """Now is the time for all good men
.....> to come to the aid of their party"""
larch> print long_string
Now is the time for all good men
to come to the aid of their party
```

It is important to keep in mind that mixing data types in a calculation may or may not make sense to Larch. For example, a string cannot be added to a integer:

```
larch> '1' + 1
Runtime Error:
cannot concatenate 'str' and 'int' objects
<StdInput>
     '1' + 1
```

but you can add an integer and a float:

```
larch> 1 + 2.5
3.5
```

and you can multiply a string by an integer:

```
larch> 'string' * 2
'stringstring'
```

Larch has special variables for boolean or logical operations: `True` and `False`. These are equal to 1 and 0, respectively, but are mostly used in logical operations, which include operators 'and', 'or', and 'not', as well as comparison operators '>', '>=', '<', '<=', '==', '!=', and 'is'. Note that 'is' expresses identity, which is a slightly stricter test than '==' (equality), and is most useful for complex objects.:

```
larch> 2 > 3
False
larch> (b > 0) and (b <= 10)
True
```

The special value `None` is used as a null value throughout Larch and Python.

Finally, Larch knows about complex numbers, using a 'j' to indicate the imaginary part of the number:

```
larch> sqrt(-1)
Warning: invalid value encountered in sqrt
nan
```

```
larch> sqrt(-1+0j)
1j
larch> 1j*1j
(-1+0j)
larch> x = sin(1+1j)
larch> print x
(1.2984575814159773+0.63496391478473613j)
larch> print x.imag
0.63496391478473613
```

To be clear, all these primitive data types in Larch are derived from the corresponding Python objects, so you can consult python documentation for further details and notes.

2.2.2 Objects and Groups

Since Larch is built upon Python, an object-oriented programming language, all named quantities or **variables** in Larch are python objects. Because of this, most Larch variables come with built-in functionality derived from their python objects. Though Larch does not provide a way for the user to define their own new objects, this can be done with the Python interface.

Objects

All Larch variables are Python objects, and so have a well-defined **type** and a set of **attributes** and **methods** that go with it. To see the Python type of any variable, use the builtin `type()` function:

```
larch> type(1)
<type 'int'>
larch> type('1')
<type 'str'>
larch> type(1.0)
<type 'float'>
larch> type(1+0j)
<type 'complex'>
larch> type(sin)
<type 'numpy.ufunc'>
```

The attributes and methods differ for each type of object, but are all accessed the same way – with a '.' (dot) separating the variable name or value from the name of the attribute or method. As above, complex data have `real` and `imag` attributes for the real and imaginary parts, which can be accessed:

```
larch> x = sin(1+1j)
larch> print x
(1.2984575814159773+0.63496391478473613j)
larch> print x.imag
0.63496391478473613
```

Methods are functions that belong to an object (and so know about the data in that object). They are also objects themselves (and so have attributes and methods), but can be called using parentheses '()', possibly with arguments inside the parentheses to change the methods behavior. For example, a complex number has a `conjugate()` method:

```
larch> x.conjugate
<built-in method conjugate of complex object at 0x178e54b8>
larch> x.conjugate()
(1.2984575814159773-0.63496391478473613j)
```

Strings and other data types have many more attributes and methods, as we'll see below.

To get a listing of all the attributes and methods of a object, use the builtin `dir()` function:

```
larch> dir(1)
['__abs__', '__add__', '__and__', '__class__', '__cmp__', '__coerce__', '__delattr__', '__div__', '__
larch> dir('a string')
['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__', '__format__', '__ge__',
```

Again, we'll see properties of objects below, as we look into more interesting data types, or you can look into Python documentation.

Groups

In addition to using basic Python objects, Larch organizes data into Groups. A Group is simply a named container for variables of any kind, including other Groups. In this way, Groups have a heirarchical structure, much like a directory of files. Each Larch variable belongs to a Group, and can be accessed by its full Group name. The top-level Group is called '_main'. You'll rarely need to use that, but it's there:

```
larch> myvar = 22.13
larch> print _main.myvar
22.13
larch> print myvar
22.13
```

You can create your own groups and add data to it with the builtin `group()` function:

```
larch> g = group()
larch> g
<Group>
```

You can add variables to your Group 'g', using the '.' (dot) to separate the parent group from the child object:

```
larch> g.x = 1002.8
larch> g.name = 'here is a string'
larch> g.data = arange(100)
larch> print g.x/5
200.56
```

(`arange()` is a builtin function to create an array of numbers). As from the above discussion of objects, the '.' (dot) notation implies that 'x', 'name', and 'data' are attributes of 'g' – that's entirely correct. Groups have no other properties than the data attributes (and functions) you add to them. Since they're objects, you can use the `dir()` function as above:

```
larch> dir(g)
['data', 'name', 'x']
```

(Note that the order shown may vary). You can also use the builtin `show()` function to get a slightly more complete view of the group's contents:

```
larch> show(g)
== Group: 3 symbols ==
  data: array<shape=(100,), type=dtype('int32')>
  name: 'here is a string'
  x: 1002.8
```

The `group()` function can take arguments of attribute names and values, so that this group could have been created with a single call:

```
larch> g = group(x=1002.8, name='here is a string', data=arange(100))
```

Many Larch functions will return groups or take a 'group' argument to write data into. That is, a function that reads data from a file will almost certainly organize that data into a group, and simply return the group for you to name, perhaps something like:

```
larch> cu = read_ascii('cu_150k.xmu')
```

Builtin Larch Groups

Larch starts up with several groups, organizing builtin functionality into different groups. The top-level '_main' group begins with 3 principle subgroups, '_builtin', '_sys', and '_math' for basic functionality. For almost all uses of Larch, several additional groups are created for more specific functionality are created on startup by Larch plugins. The principle starting groups are describe in *Table of Basic Larch Groups*

Table of Basic Larch Groups. These groups are listed in order of how they will be searched for functions and data.

Group Name	description
_builtin	basic builtin functions.
_math	mathematical and array functions.
_sys	larch sstem-wide variables.
_io	file input/output functions.
_plotter	plotting and image display functions.
_xafs	XAFS-specific functions.

The functions in '_builtin' are mostly inherited from Python's own built-in functions. The functions in '_math' are mostly inherited from Numpy, and contain basic array handling and math.

How Larch finds variable names

With several builtin groups, and even more groups created to store your own data to be processed, Larch ends up with a complex heirarchy of data. This gives a good way of organizing data, but it also leads to a question of how variable names are found. Of course, you can always access a function or data object by its full name:

```
larch> print _math.sin(_math.pi/2)
1.0
```

but that's too painful to use, and of course, one needs to be able to do:

```
larch> print sin(pi/2)
1.0
```

and have Larch know that when you say `sin()`, you mean `_math.sin()`. The way this look-up of names works is that Larch keeps a list of groups that it will search through for names. This list is held in the variable `_sys.searchGroups`, and can be viewed and modified during a Lach session. On startup, this list has the groups listed in *Table of Basic Larch Groups*, in the order shown. To be clear, if there was a variable named `_sys.something` and a `_math.something`, typing 'something' would resolve to `_sys.something`, and to access `_math.something` you would have to give the full name. For the builtin functions and variables, such clashes are not so likely, but they are likely if you read in many data sets as groups, and want to access the contents of the different groups.

2.2.3 More Complex Data Structures: Lists, Arrays, Dictionaries

Larch has many more data types built on top of the primitive types above. These are generally useful for storing collections of data, and can be built up to construct very complex structures. These are all described in some detail

here. But as these are all closely related to Python objects, further details can be found in the standard Python documentation.

Lists

A list is an ordered sequence of other data types. They are **heterogeneous** – they can be made up of data with different types. A list is constructed using brackets, with commas to separate the individual:

```
larch> my_list1 = [1, 2, 3]
larch> my_list2 = [1, 'string', sqrt(7)]
```

A list can contain a list as one of its elements:

```
larch> nested_list = ['a', 'b', ['c', 'd', ['e', 'f', 'g']]]
```

You can access the elements of a list using brackets and the integer index (starting from 0):

```
larch> print my_list2[1]
'string'
larch> print nested_list[2]
['c', 'd', ['e', 'f', 'g']]
larch> print nested_list[2][0]
'c'
```

Lists are **mutable** – they can be changed, in place. To do this, you can replace an element in a list:

```
larch> my_list1[0] = 'hello'
larch> my_list1
['hello', 2, 3]
```

As above, lists are python **objects**, and so come with methods for interacting with them. For example, you can also change a list by appending to it with the 'append' method:

```
larch> my_list1.append('number 4, the larch')
larch> my_list1
['hello', 2, 3, 'number 4, the larch']
```

All lists will have an 'append' method, as well as several others:

- count – to return the number of times a particular element occurss in the list
- extend – to extend a list with another list
- index – to find the first occurance of an element
- insert – to insert an element in a particular place.
- pop – to remove and return the last element (or other specified index).
- remove – remove a particular element
- reverse – reverse the order of elements
- sort – sort the elements.

Note that the methods that change the list do so *IN PLACE* and return `None`. That is, to sort a list, do this:

```
larch> my_list.sort()
```

but not this:

```
larch> my_list = my_list.sort()   # WRONG!!
```

as that will set 'my_list' to None.

You can get the length of a list with the built-in `len()` function, and test whether a particular element is in a list with the *in* operator:

```
larch> my_list = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']
larch> print len(my_list)
10
larch> 'e' in my_list
True
```

You can access a sub-selection of elements with a **slice**, giving starting and ending indices between brackets, separated by a colon. Of course, the counting for a slice starts at 0. It also excludesthe final index:

```
larch> my_list[1:3]
['b', 'c']
larch> my_list[:4]    # Note implied 0!
['a', 'b', 'c', 'd']
```

You can count backwards, and using '-1' is a convenient way to get the last element of a list. You can also add an optional third value to the slice for a step:

```
larch> my_list[-1]
'j'
larch> my_list[-3:]
['h', 'i', 'j']
larch> my_list[::2]  # every other element, starting at 0
['a', 'c', 'e', 'g', 'i']
larch> my_list[1::2]  # every other element, starting at 1
['b', 'd', 'f', 'h', 'j']
```

A final important property of lists, and of basic variable creation in Larch (and Python) is related to the discussion above about variable creation and assignment. There we said that 'creating a variable':

```
larch> my_list = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']
```

was best thought of as creating a value (here, the literal list "['a', 'b', ..., 'j']") and then assigning the name 'my_list' to point to that value. Here's why we make that distinction. If you now say:

```
larch> your_list = my_list
```

the variable 'your_list' now points to the same value – the same list. That is, it does not make a copy of the list. Since the list is mutable, changing 'your_list' will also change 'my_list':

```
larch> your_list[0] = 500
larch> print my_list[:3]
[500, 'b', 'c']                # changed!!
```

You can make a copy of a list, by selecting a full slice:

```
larch> your_list = my_list[:]
larch> your_list[0] = 3.2444
larch> print my_list[:3]
[500, 'b', 'c']                 # now unchanged

larch> your_list[0] == my_list[0]
False
```

Note that this behavior doesn't happen for immutable data types, including the more primitive data types such as integers, floats and strings. This is essentially because you cannot assign to parts of those data types, only set its entire value.

As always, consult the Python documentation for more details.

Tuples

Like lists, tuples are sequences of heterogenous objects. The principle difference is that tuples are **immutable** – they cannot be changed once they are created. Instead, tuples are a simple ordered container of data. The syntax for tuples uses comma separated values inside (optional!) parentheses in place of brackets:

```
larch> my_tuple = (1, 'H', 'hydrogen')
```

Like lists, tuples can be indexed and sliced:

```
larch> my_tuple[:2]
(1, 'H')
larch> my_tuple[-1]
'hydrogen'
```

Due to their immutability, tuples have only a few methods ('count' and 'index' with similar functionality as for list).

Though tuples they may seem less powerful than lists, and they are actually used widely with Larch and Python. In addition to the example above using a tuple for a short, fixed data structure, many functions will return a tuple of values. For this case, the simplicity an immutability of tuples is a strength becaues, once created, a tuple has a predictable size and order to its elements, which is not true for lists. That is, if a larch procedure (which we'll see more of below) returns two values as a tuple:

```
larch> def sumdiff(x, y):
.....>     return x+y, x-y
.....> enddef
larch> x = sumdiff(3, 2)
larch> print x[0], x[1]
5 1
```

Because the returned tuple has a fixed structure, you can also assign the it directly to a set of (the correct number of) variables:

```
larch> plus, minus = sumdiff(10, 3)
larch> print plus, minus
13 7
```

A second look at Strings

Though discussed earlier in the basic data types, strings are closely related to lists as well – they are best thought of as a sequence of characters. Like tuples, strings are actually immutable, in that you cannot change part of a string, instead you must create a new string. Strings can be indexed and sliced as with lists and tuples:

```
larch> name = 'Montaigne'
larch> name[:4]
'Mont'
```

Strings have many methods – over 30 of them, in fact. To convert a string to lower case, use its `lower()` method, and so on:

```
larch> 'Here is a String'.lower()
'here is a string'
larch> 'Here is a String'.upper()
'HERE IS A STRING'
larch> 'Here is a String'.title()
'Here Is A String'
```

This aslo shows that the methods are associated with strings themselves – even literal strings, and simply with variable names.

Strings can be split into words with the `split()` method, which splits a string on whitespace by default, but can take an argument to change the character (or substring) to use to split the string:

```
larch> 'Here is a String'.split()
['Here', 'is', 'a', 'String']

larch> 'Here is a String'.split('i')
['Here ', 's a Str', 'ng']
```

As above, this is really only touching the tip of the iceberg of string functionality, and consulting standard Python documentation is recommended for more information.

Arrays

Whereas lists are sequences of heterogeneous objects that can grow and shrink, and included deeply nested structures, they are not well suited for holding numerical data. Arrays are sequences of the same primitive data type, and so are much closer to arrays in C or Fortran. This makes them much more suitable for numeric calculations, and so are extremely important in Larch. There are many ways to create arrays, including the builtin `array()` function which will attempt to convert a list or tuple of numbers into an Array. You can also use the builtin `arange()` function to create an ordered sequence of indices ([1, 2, 3, ...]), and several other methods listed in

Table of Array Creation Functions

Table of Array Creation Functions. These functions can all be used to create arrays in Larch.

Function Name	description	example
array	array from list	arr = array([1,2,3])
arange	indices 0, N-1	arr = arange(10)
zeros	fill with N zeros	arr = zeros(10)
ones	fill with N ones	arr = ones(10)
linspace	fill with bounds and N	arra = linspace(0, 1, 11)

Some examples of using these functions are needed:

```
larch> i = arange(10)
larch> i
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
larch> f = arange(10, dtype='f8')
larch> f
array([0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])
larch> c = arange(10, dtype='c16')
larch> c
array([ 0.+0.j,  1.+0.j,  2.+0.j,  3.+0.j,  4.+0.j,  5.+0.j,  6.+0.j,
        7.+0.j,  8.+0.j,  9.+0.j])
```

Here, the **dtype** argument sets the data type for the array members – in this case 'f8' means '8 byte floating point' and 'c16' means '16 byte complex' (i.e, double precision, and double precision complex, respectively).

The `linspace()` function is particularly useful for creating arrays, as it takes a starting value, ending value, and number of points between these:

```
larch> s = linspace(0, 10, 21)
larch> s
array([  0. ,   0.5,   1. ,   1.5,   2. ,   2.5,   3. ,   3.5,   4. ,
         4.5,   5. ,   5.5,   6. ,   6.5,   7. ,   7.5,   8. ,   8.5,
         9. ,   9.5,  10. ])
```

Several variants are possible. For more information, consult the numpy tutorials, or use the online help system within Larch (which will print out the documentation string from the underlying numpy function):

```
larch> help(linspace)

    Return evenly spaced numbers over a specified interval.

    Returns `num` evenly spaced samples, calculated over the
    interval [`start`, `stop` ].

    The endpoint of the interval can optionally be excluded.

    Parameters
    ----------
    start : scalar
        The starting value of the sequence.
    stop : scalar
        The end value of the sequence, unless `endpoint` is set to False.
        In that case, the sequence consists of all but the last of ``num + 1``
        evenly spaced samples, so that `stop` is excluded.  Note that the step
        size changes when `endpoint` is False.
    num : int, optional
        Number of samples to generate. Default is 50.
    endpoint : bool, optional
        If True, `stop` is the last sample. Otherwise, it is not included.
        Default is True.
    retstep : bool, optional
        If True, return (`samples`, `step`), where `step` is the spacing
        between samples.

    ....
```

Dictionaries

Our final basic data-structure is the dictionary, which is a container that maps values to keys. This is sometimes called a hash or associative array. Like a list, a dictionary holds many heterogeneous values, and can be altered in place. Unlike a list, the elements of a dictionary have no guaranteed order, and are not selected by integer index, and multiple values cannot be selected by a slice. Instead, the elements of a dictionary are accessed by key, which is normally a string, but can also be an integer or floating point number, or even a tuple or some other objects – any **immutable** object can be used. Dictionaries are delimited by curly braces, with colons (':') separating key and value, and commas separating different elements:

```
larch> atomic_weight = {'H': 1.008, 'He': 4.0026, 'Li': 6.9, 'Be': 9.012}
larch> print atomic_weight['He']
4.0026
```

You can also add more elements to a dictionary by assigning to a new key:

```
larch> atomic_weight['B']  = 10.811
larch> atomic_weight['C']  = 12.01
```

Dictionaries have several methods, such as to return all the keys or all the values, with:

```
larch> atomic_weight.keys()
['Be', 'C', 'B', 'H', 'Li', 'He']
larch> atomic_weight.values()
[9.0120000000000005, 12.01, 10.811, 1.008, 6.9000000000000004, 4.0026000000000002]
```

Note that the keys and values are not in the order they were entered in, but do have the same order.

As with lists, dictionaries are mutable, and the values in a dictionary can be any object, including other lists and dictionaries, so that a dictionary can end up with a very complex structure. Dictionaries are quite useful, and are in fact used throughout python.

2.3 Tutorial: Conditional Execution and Flow Control

Two important needs for a full-featured language are the ability to run different statements under different conditions, and to repeat certain calculations. These are generally called 'flow control', as these statements control how the program will flow through the text of the script. In the discussion here, we will also introduce a few new concepts and Larch statements.

So far in this tutorial, all the text written to the Larch command line has been a single line of text that is immediately run, either printing output to the terminal or assigning a value to a variable. These are both examples of **statements**, which are the basic pieces pf text you send to the program. So far we've seen three types of statements:

1. simple statements or expressions, such as:

   ```
   larch> 1+sqrt(3)
   ```

   or:

   ```
   larch> atomic_weight.keys()
   ```

   where we have an expression evaluated. At the command line, these values are printed – in a script they would not be printed.

2. print statements, such as:

   ```
   larch> print sqrt(3)
   ```

   where we explicitly command larch to print the evaluated expression – this would print if run from a script.

3. assignment statements, such as:

   ```
   larch> x = sqrt(3)
   ```

   where we assign the name 'x' to hold the evaluated expression.

In fact, though these are the most common types of statements, there are many more types of statements. We will introduce a few more statement types here, including compound statements that take up more than one line of test.

2.3.1 Conditional Evaluation with if and else

A fundamental need is to be able to execute some statement(s) when some condition is met. This is done with the **if** statement, an example of which is:

```
larch> if x == 0: print 'x is 0!'
```

Which will print 'x is 0!' if the value of *x* is equal to 0. The *x == 0* in this if statement is called the **test**. A test is a Boolean mathematical expression. While most usual mathematical expressions use operators such as '+', '-', a Boolean expression uses the operators listed in *Table of Boolean Operators* to evaluate to a value of `True` or `False`.

A single-line if statement as above looks like this:: if <test>: statement

The 'if' and ':' are important, while '<test>' can be any Boolean expression. If the test evaluates to `True`, the statement is executed.

If statements can execute multiple statements by putting the statements into a "block of code":

```
if x == 0:
    print 'x is equal to 0!'
    x = 1
endif
```

Which is to say that the multiple-line form of the if statement looks like this:

```
if <test>:
  <statements>
endif
```

where '<statements>' here means a list of statements, and the 'endif' is required (see *Code Block Ends*). For the above, two statements will be run if x is equal to 0 – there is no restriction on how many statements can be run.

An 'else' statement can be added to execute code if the test is False:

```
if x == 0:
    print 'x is equal to 0!'
    x = 1
else
    print 'x is not 0'
endif
```

Multiple tests can be chained together with the 'elif' (a contraction of 'else if'):

```
if x == 0:
    print 'x is equal to 0!'
elif x > 0:
    print 'x is positive'
elif x > -10:
    print 'x is a small negative number'
else:
    print 'x is negative'
endif
```

Here the 'x > 0' test will be executed if the 'x == 0' test fails, and the 'x > -10' test will be tried if that fails. **Table of Boolean Operators** The operators here all take the form *right OP left* where OP is one of the operators below. Note the distinction between '==' and 'is'. The former compares *values* while the latter compares the identity of two objects.

boolean operator	meaning
==	has equal value
!=	has unequal value
>	has greater value
>=	has greater or equal value
<	has smaller value
<=	has smaller or equal value
is	is identical to
not	is not `True`
and	both operands are `True`
or	either operand is `True`

Note that in Larch, as in Python, any value can be used as a test, not just values that are `True` or `False`. As you might expect, for example, the value 0 is treated as `False`. An empty string is also treated as `False`, as is an empty list or dictionary. Most other values are interpreted as `True`.

2.3.2 For loops

It is often necessary to repeat a calculation multiple times. A common method of doing this is to use a **loop**, including using a loop counter to iterates over some set of values. In Larch, this is done with a **for loop**. For those familiar with other languages, a Larch for loop is a bit different from a C for loop or Fortran do loop. A for loop in Larch iterates over an ordered set of values as from a list, tuple, or array, or over the keys from a dictionary. Thus a loop like this:

```
for x in ('a', 'b', 'c'):
    print x
endfor
```

will go through values 'a', 'b', and 'c', assigning each value to *x*, then printing the value of x, which will result in printing out:

```
a
b
c
```

Similar to the the *if* statement above, the for loop has the form:

```
for <varlist> in <sequence>:
    <statements>
endfor
```

Compared to a C for loop or Fortran do loop, the Larch for loop is much more like a *foreach* loop. The common C / Fortran use case of interating over a set of integers can be emulated using the builtin `range()` function which generates a sequence of integers. Thus:

```
for i in range(5):
   print i, i/2.0
endfor
```

will result in:

```
0, 0.0
1, 0.5
2, 1.0
3, 1.5
4, 2.0
```

Note that the builtin `range()` function generates a sequence of integers, and can take more than 1 argument to indicate a starting value and step. It is important to note that the sequence that is iterated order does not be generated

from the `range()` function, but can be any list, array, or Python sequence. Importantly, this includes strings(!) so that:

```
for char in 'hello':  print char
```

will print:

```
h
e
l
l
o
```

This can cause a common sort of error, in that you might expect some variabe to hold a list of string values, but it actually holds a single string. Notice that:

```
filelist = ('file1', 'file2')
for fname in filelist:
    fh = open(fname)
    process_file(fh)
    fh.close()
endfor
```

would act very differently if filelist was changed to 'file1'!

Multiple values can be assigned in each iteration of the for loop. Thus, iterating over a sequence of equal-length tuples, as in:

```
for a, b in (('a', 1), ('b', 2), ('c', 3)):
    print a, b
endfor
```

will print:

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

This may seem to be mostly of curious interest, but can be extremely useful especially when dealing with dictionaries or with arrays or lists of equal length. For a dictionary *d*, *d.items()* will return a list of two-element tuples as above of key, value. Thus:

```
mydict = {'a':1, 'b':2, 'c':3, 'd':4}
for key, val in mydict.items():
    print key, val
endfor
```

will print (note that dictionaries do no preserve order, but the (key, val) pairs match:

```
a 1
c 3
b 2
d 4
```

The builtin `zip()` function is similarly useful, turning a sequence of lists or arrays into a sequence of tuples of the corresponding elements of the lists or arrays. Thus:

```
larch> a = range(10)
larch> b = sin(a)
larch> c = cos(a)
larch> print zip(a, b, c)
```

```
[(0, 0.0, 1.0), (1, 0.8414709848078965, 0.54030230586813977),
 (2, 0.90929742682568171, -0.41614683654714241), ....]
```

(Note that for arrays or lists of unequal length, `zip()` will return tuples until any of its arguments runs out of elements). Thus a for loop can make use of the `zip()` function to iterate over multiple arrays:

```
larch> a = arange(101)/10.0
larch> print 'X   SIN(X)  SIN(Y)\n================\n'
larch> for a, sval, cval in zip(a, sin(a), cos(a)):
.....>     print '%.3f, %.5f, %.5f' % (a, sval, cval)
.....> endfor
```

will print a table of sine and cosine values.

A final utility of note for loops is `enumerate()` which will return a tuple of (index, value) for a sequence. That is:

```
larch> for i, a in ('a', b', 'c'):
.....>     print i, a
.....> endfor
```

will print:

```
0, a
1, b
2, c
```

It is sometimes useful to jump out of a for loop, or go onto the next value in the sequence. The *break* statement will exit a for loop immediately:

```
for fname in filelist:
    status = get_status(fname)
    if status < 0:
       break
    endif
    more_processing(fname)
endfor
print 'processed up to i = ', i
```

may jump out of the loop before the sequence generated by 'range(10)' is complete. The variable 'i' will have the final used value.

To skipover an iteration of a loop but continue on, use the *continue* statement:

```
for fname in filelist:
    status = get_status(fname)
    if status < 0:
       continue
    endif
    more_processing(fname)
endfor
```

2.3.3 While loops

While a for loop generally walks through a pre-defined set of values, a *while* loop executes as long as some test is `True`. The basic form is:

```
while <test>:
   <statements>
endwhile
```

Here, the test works as for *if* – it is a Boolean expression, evaluated at each iteration of the loop. Generally, the expression will test something that has been changed inside the loop (even if implicitly). The classic while loop increases a counter at each iteration:

```
counter = 0
while counter < 10:
   do_something(counter)
   counter =+ 1
endwhile
```

A while loop is easily turned into an infinite loop, simply by not incrementing the counter. Then again, the above loop would easily be converted into a for loop, as the counter is incremented by a fixed amout at each iteration. A more realistic use would be:

```
n = 1
while n < 100:
   n = (n + 0.1) * n
   print n
endwhile
```

An additional use for a while loop is to use an implicit or external condition, such as time:

```
now = time.time() # return the time in seconds since Unix epoch
while time.time() - now < 15:   # That is 'for 15 seconds'
   do_someting()
endwhile
```

The *break* and *continue* statements also work for while loops, just as they do with for loops. These can be used as ways to exit an other-wise infinite while loop:

```
while True:  # will never exit without break!
   answer = raw_input('guess my favorite color>')
   if answer == 'lime':
       break
   else:
       print 'Nope, try again'
   endif
endwhile
```

2.4 Tutorial: Dealing With Errors

When an error exists in the syntax of your script, or an erro happens when running your script, an *Error* or *Exception* is generated, and the execution of your script is stopped.

2.4.1 Syntax Errors

Syntax errors result from incomplete or ill-formed larch syntax. For example:

```
larch> x = 3 *
SyntaxError
<StdInput>
    x = 3  *
```

This indicates that the Larch interpreter could not understand the meaning of the statement 'x = 3 *', because it excepts a value after the *''*. Syntax errors are spotted and raised before the interpreter tries to evaluate the expression. That is because Larch first fully parses any statements (or block of statements if you're entering multiple statements or loading

a script from a file) into a partially compiled, executable form before executing. Because of this two-step approach (first parse to intermediate form, then execute that intermediate form), syntax errors are sometimes referred to as a parsing errors.

2.4.2 Exceptions

Even if the syntax of your script is correct, the logic might not be. In addition, even if the logic is correct for most cases, it might not be correct for all. For example, certain values might cause an error run time:

```
larch> n = 1
larch> print 4.0 / ( n - 1)
ZeroDivisionError('float division')
<StdInput>
    print 4.0/(n-1)
          ^^^
```

which is saying that you can't divide by 0. This is known as a **Runtime Exception**. It might indicate a programming error, for example that you didn't test if the denominator was 0 before doing the division.

Larch (as inherited from Python) has many different types of exceptions, so that dividing by zero, as above, is detected as a different exception from, say, trying to open a file that doesn't exist:

```
larch> fh = open('foo', 'r')
Error running <built-in function open>
IOError(2, 'No such file or directory')
<StdInput>
    fh = open('foo', 'r')
         ^^^
```

or trying to add an integer and a string together:

```
larch> 4 + 'a'
TypeError("unsupported operand type(s) for +: 'int' and 'str'")
<StdInput>
    4 + 'a'
```

Though they are called exceptions, such problems are fairly common when developing programs or writing scripts. Having a built-in way to test for and handle different kinds of exceptions is an important part of many modern computer languages, and Larch has this capability with its **try** and **except** statements.

2.4.3 Try and Except

The **try** statement will execute a block of code and look for certain types of exceptions. One or more **except** statements can be added to specify blocks of code to execute if the specified exception occurs. As a simple example:

```
try:
    x = a/b
except ZeroDivisionError:
    print 'saw a divide by zero!
    x = 0
endtry
<more statements>
```

If b is not 0, x is set to the value of a/b. If b is 0, executing *x = a/b* will cause a ZeroDivisionError (as we saw above), so the block with the print statement and setting x to 0 will be executed. In either of these cases (no exception, or a handled exception), execution will continue as normal. If a different problem occurs – an "unhandled exception"

– such as the case if *a* holds a string value with *b* holds an integer, then execution will stop and the corresponding exception will be raised.

There can be several **except** statements for each **try** statement, to check for multiple types of problems. These will be checked in order. For example:

```
try:
    x = a/b
except ZeroDivisionError:
    print 'saw a divide by zero!
    x = 0
except TypeError:
    print "a and b are of different types -- can't divide"
endtry
<more statements>
```

It is sometimes useful to run certain code only when a looked-for error has not occurred. For example, it is often agood idea to test when opening a file for IOError (which covers a range of issues such as the file not being found), and only reading that file if it actually opened. For example, to read in a file into a list of lines, the recommended practice is to do:

```
try:
    fh = open(filename, 'r')
except IOError:
    print 'cannot open file %s!' % filename
    datalines = []
else:
    datalines = fh.readlines()
    fh.close()
endtry
<operate on datalines>
```

There is a very large number of exception types built into Larch, all inherited from Python. See the standard Python documentation for more details.

2.4.4 Raising your own exceptions

In certain cases, you may want to cause an exception to occur. This need is most likely to happen when writing your own procedures, and want to ensure that the input arguments can be handled correctly.

To cause an exception, you use the **raise** statement, and you are said to be "raising an exception":

```
larch> raise TypeError("wrong data type")
```

2.5 Tutorial: Writing Procedures

Any moderately complex script will eventually include calcluations that need to be repeated. The preferred way to do this is write your own function or **procedure**, which can be called exactly as the built-in functions. For clarification, here we use the word **procedure** for a function written in Larch, leaving **function** to imply a Python function. In fact, as we will see, there is very little difference in practical use.

Once you're ready to write procedures, you'll almost certainly want to read about running Larch scripts and modules in the next section of this tutorial.

2.5.1 Def statement

To define a procedure, you use the **def** statement, and write a block of code. This looks much like the **if**, **for**, and **while** blocks discussed earlier. A simple example would be:

```
def sayhello():
    print 'hello!'
enddef
```

With this definition, one can then run this procedure as you would run any other built-in function, by writing:

```
larch> sayhello()
hello!
```

Of course, you can write procedures that take input arguments, such as:

```
def safe_sqrt(x):
  if x > 0:
     print sqrt(x)
  else:
     print 'Did you want sqrt(%f) = %f?' % (-x,  sqrt(-x))
  endif
enddef
```

Here *x* will hold whatever value is passed to it, so that:

```
larch> safe_sqrt(4)
2.0
larch> safe_sqrt(-9)
Did you want sqrt(9.000000) = 3.000000?
```

Of course, you will most often want a procedure to return a value. This is done with the **return** statement. This can be put anywhere in a procedure definition. When encountered, it will cause the procedure to immediately exist, passing back any indicated value(s). If no **return** statement is given in a procedure, it will return `None` when the procedure has fully executed. An example:

```
def safe_sqrt(x):
  if x > 0:
     return sqrt(x)
  else:
     return sqrt(-x)
  endif
enddef
```

which can no be used as:

```
larch> print safe_sqrt(4)
2.0
larch> x = safe_sqrt(-10)
larch> print x
3.16227766017
```

Multiple arguments can be specified, separated by a comma:

```
larch> def my_add(x, y):
.....>    return x + y
.....> enddef
larch> print my_add(1, 2.2)
3.2
```

Formally, the definition of a procedure looks like:

```
def <procedure_name>(<arguments>):
    <block of statements>
enddef
```

2.5.2 Namespace and "Scope" inside a Procedure

While inside a procedure, and important consideration is "what variables does this procedure have access to?". Generally speaking, there is no reason to expect it to know about any variables that are not passed in as arguments or created inside the procedure definition.

2.5.3 The return statement, and multiple Return values

As seen above, the **return** statement will exist a procedure, and send back a value to the calling code. The return value can be either a single value or a tuple of values, which gives a convenient way to return multiple values from a single procedure. Thus:

```
larch> def my_divmod(x, y):
.....>    return (x // y, x % y)  # note use of // for integer division!
.....> enddef
larch> print my_divmod(100, 7)
14, 2
```

But be careful when assigning the return value to variable(s). You can do:

```
larch> xdiv, xmod = my_divmod(100, 7)
larch> print xdiv
14
```

or:

```
larch> result = my_divmod(100, 7)
larch> print result[0], result[1]
14, 2
```

Because a return value from a procedure can hold many values, it is best to be careful when writing a procedure that you document what the return value is, and when using a procedure that you're getting the correct number of values.

2.5.4 Keyword arguments

For the procedures defined so far, the arguments have been both required and in a fixed order. Sometimes, you'll want to give a procedure optional arguments, and perhaps allow some flexibility in the order of the arguments. Larch allows this with **keyword** arguments. In a procedure definition, you add an argument name with a default value, like this:

```
def xlog(a, base=e):
    """return log(a) with base = base (default=e=2.71828...)
    """
     if base > 1:
        return log(a) /log(base)
    else:
        print 'cannot calculate log base %f' % base
    endif
enddef
```

Unless passed in, the value of *base* will take the default value of *e*. This can then be used as:

```
larch> xlog(16)
2.7725887222397811
larch> xlog(16, base=10)
1.2041199826559246
larch> xlog(16, base=2)
4.0
```

You can supply many keyword arguments, but they must all come *after* the positional arguments.

A procedure can be written to take an unspecified number of positional and keyword parameters, using a special syntax for unspecified positional arguments and for unspecified keyword arguments. To use unspecified positional arguments, a procedure definition takes an argument preceded by a '*' after all the named positional arguments, like this:

```
def addall(a, b, *args):
    """add all (at least 2!!) arguments given"""
    out = a + b
    for c in args:
        out = out + c
    endfor
    return out
enddef
```

Here, the **'*args'** arguments means to use the variable 'args' to hold any number of positional arguments beyond those explicitly given. Inside the procedure, a tuple named 'args' will hold any positional parameters included in the call to 'addall' past the first two (which will be held by 'a' and 'b'). Thus, this procedure can be used as:

```
larch> addall(2, 3)          # args = ()
5
larch> addall(2, 3, 5, 7)    # args = (5, 7)
17
```

To add support for unspecified keyword parameters, one adds a named argument to the procedure definition preceded by two asterisks: **'**keywords'**. For example:

```
def operate(a, b, **options):
    """perform operation on a and b"""
    debug = options.get('debug', True)
    verbose = options.get('verbose', False)
    op  = options.get('op', 'add')
    if verbose:
       print 'op == %s ' % op
    endif
    if op == 'add':
        return a + b
   elif op == 'sub':
        return a - b
   elif op == 'mul':
        return a * b
   elif op == 'div':
        return a / b
   else:
        if debug:  print 'unsupported operation!'
   endif
enddef
```

As you may have figured out, inside the procedure, 'options' will hold a dictionary of keyword names/values passed into it. With this (perhaps contrived) definition, you can call 'operate' many ways to change its behavior:

```
larch> operate(3, 2, op='add')
5
larch> operate(3, 2, op='add', verbose=True)
op == add
5
larch> operate(3, 2, op='mul', verbose=True)
op == mul
6
larch> operate(3, 2, op='xxx', verbose=True)
op == xxx
unsupported operation!
larch> operate(3, 2, op='xxx', debug=False)
op == xxx
```

As with the **'*args'**, the **'**options'** in the procedure definition must appear after any named keyword parameters, and will not include the named keyword parameters.

2.5.5 Documentation Strings

Of course, it is a good idea to document your procedures so that you and others can read what it is meant to do and how to use it. Larch has a built-in mechanism for supporting procedure documentaion. If the first statement in a procedure is a **bare string** (that is, a string that is not assigned to a variable), then this will be used as the procedure documentation. You can use triple-quoted strings for multi-line documentation strings. This doc string will be used by the built-in help mechanism, or when viewing details of the procedure. For example:

```
def safe_sqrt(x):
  """safe sqrt function:
 returns sqrt(abs(x))
 """
 return sqrt(abs(x))
enddef
```

With this definition:

```
larch> help(safe_sqrt)
  safe sqrt function:
     returns sqrt(abs(x))
```

2.6 Tutorial: Running Larch Scripts, and Modules

Once you've done any significant amount of work with Larch, you'll want to save what you've done to a file of Larch code, and run it over again, perhaps changing some input. There are a few ways to do this. Writing procedures that can be re-used is a highly recommended approach.

2.6.1 Running a script with run

If you have a file of Larch code, you can run it with the built-in **run** function:

```
# file  myscript.lar
print 'hello from myscript.lar!
for i in range(5):
   print i, sqrt(a)
endfor
#
```

To run this, you simply type:

```
larch> run('myscript.lar')
hello from myscript.lar!
0 0.0
1 1.0
2 1.41421356237
3 1.73205080757
4 2.0
```

A script can contain any Larch code, including procedure definitions. After running the script, any variables assigned in the script will exist in your larch session. For example, after the loop in *myscript.lar*, the variable *i* will be 4, and you can access this variable:

```
larch> print i
4
```

This is to say that the script runs in the "top-level namespace", about which we'll see more below.

2.6.2 Importing a Larch Module

A larch script can also be **imported** using the **import** statement:

```
larch> import myscript
```

Notice a few differences. First, the '.lar' suffix was removed. Second, the name is **NOT IN QUOTES** as one my expect for a string containing a file name. This is because the **import** statement knows what extensions to look for:

- module lookup
- import as..
- from xx import yy as zz

2.6.3 Namespaces, again

2.7 Tutorial: Reading and Writing Data

Larch has several built-in functions

2.8 Tutorial: Plotting and Displaying Data

Plotting and Visualizing data is vital to a scientific analysis package, and Larch provides several methods for data visualization.

CHAPTER

THREE

LARCH REFERENCE

This chapter describes further details of Larch language, intending to act as a reference manual. As discussed elsewhere, the single most important fact about Larch is that it implemented with and closely related to Python. Of course, Python is very well documented, and much of the Python documentation can be applied to Larch. Thus the discussion here focuses on the differences betwen Larch and Python, and on the functionality unique to Larch.

Much of the discussion here will expect a familiarity with programming and a willingness to consult the on-line Python documentation when necessary.

3.1 Overview

Larch requires Python version 2.6 or higher. Support for Python 3.X is partial, in that the core of Larch does work but is not particularly well-tested. Importantly, wxPython, the principle GUI toolkit used by Larch, has not yet been ported to Python 3.X, and so no graphical or plotting capabilities are available yet for Larch using Python 3.

CHAPTER

FOUR

LARCH FOR DEVELOPERS

This chapter describes details of Larch language for developers and programmers wanting to extend Larch. This document will assume you have some familiarity with Python.

4.1 Differences between Larch and Python

Larch is very similar to Python but there are some very important diffferences, especially for someone familiar with Python. These differences are not because we feel Python is somehow inadequate or imperfect, but because Larch is a domain-specific-language. It is really something of an implementation detail that Larch's syntax is so close to Python. The principle differences with Python are:

1. Using 'end*' instead of significant white-space for code blocks.
2. Groups versus Modules
3. Changing the lookup rules for symbol names
4. Not implementing several python concepts, notably Class, and lambda.

Each of these is discussed in more detail below.

Some background and discussion of Larch implementation may help inform the discussion below, and help describe many of the design decisions made in Larch. First and foremost, Larch is designed to be a domain-specific macro language that makes heavy use of Python's wonderful tools for processing scientific data. Having a macro language that was similar to Python was not the primary goal. The first version of Larch actually had much weaker correspondance to Python. It turned out that the implementation was much easier and more robust when using syntax close to Python.

When larch code is run, the text is first *translated into Python code* and then parsed by Python's own *ast* module which parses Python code into an *abstract syntax tree* that is much more convenient for a machine to execute. As a brief description of what this module does, the statement:

```
a*sin(2*b)+c
```

will be parsed and translated into something like:

```
Add(Name('c'), Mult(Name('a'), Call(Name('sin'), Args([Mult(Num(2), Name('b'))]))))
```

Larch then walks through this tree, executing each Add, Mult, etc on its arguments. If you've ever done any text processing or thought about how a compiler works, you'll see that having this translation step done by proven tools is a huge benefit. For one thing, using Python's own interpreter means that Larch simply does not having parsing errors – any problem would be translation of Larch code into Python, or in executing the compiled code above. This also makes the core code implementing Larch much easier (the core functionality is fewer than 3000 lines of code).

Given this main implementation feature of Larch, you can probably see where and how the differences with Python arise:

- The Larch-to-Python translation step converts the 'end*' keywords into significant whitespace ('commenting out 'endif' etc if needed).
- The lookup for symbols in **Name('c')** is done at run-time, allowing changes from the standard Python name lookup rules.
- Unimplemented Python constructs (class, lambda, etc) are parsed, but

You can also see that Python's syntax is followed very closely, so that the translation from Larch-to-Python is minimal.

4.1.1 Code Block Ends

Unlike Python, Larch does not use significant whitespace to define blocks. There, that was easy. Instead, Larch uses "end blocks", of the form:

```
if test:
   <block of statements>
endif
```

Each of the Keywords *if*, *while*, *for*, *def*, and *try* must be matched with a corresponding 'end' keyword: *endif*, *endwhile*, *endfor*, *enddef*, and *endtry*. You do not need an *endelse*, *endelif*, *endexcept*, etc, as this is not ambiguous.

As a special note, you can place a '#' in front of 'end'. Note that this means exactly 1 '#' and exactly in front of 'end', so that '#endif' is allowed but not '####endif' or '# endfor'. This allows you to follow Python's indenting rules and write code that is valid Larch and valid Python, which can be useful in translating code:

```
for i in range(5)
    print(i, i/2.0)
#endfor
```

This code is both valid Larch and valid Python.

4.1.2 Groups vs Modules

This is at least partly a semantic distinction. Larch organizes data and code into Groups – simple containers that hold data, functions, and other groups. These are implemented as a simple, empty class that is part of the symbol table.

4.1.3 Symbol Lookup Rules

The name lookups in Python are quite straightforward and strict: local, module, global. They are also fairly focused on *code* rather than *data*. There is a tendency with Python scripts to use something like:

```
from numpy import *
```

in quick-and-dirty scripts, though many experienced developers will tell you to avoid this like the plague.

In Larch, there is a list of Groups namespaces that are

4.1.4 Unimplemented features

A domain-specific-language like Larch does not need to be as full-featured as Python, so we left a few things out. These include (this may not be an exhaustive list):

- eval – Larch *is* sort of a Python eval
- lambda

- class
- global
- generators, yield
- decorators

4.2 Modules

Larch can import modules either written in Larch (with a '.lar' extension) or Python (with a '.py' extension). When importing a Python module, the full set of Python objects is imported as a module, which looks and acts exactly like a Group.

4.3 Plugins

Plugins are a powerful feature of Larch that allow it to be easily extended without the requiring detailed knowledge of all of Larch's internals. A plugin is a specially written Python module that is meant to add functionality to Larch at run-time. Generally speaking, plugins will be python modules which define new or customized versions of functions to create or manipulate Groups.

Plugins need access to Larch's symbol table and need to tell Larch how to use them. To do this, each function to be added to Larch in a plugin module needs a *larch* keyword argument, which will be used to pass in the instance of the current larch interpreter. Normally, you will only need the *symtable* attribute of the *larch* variable, which is the symbol table used.

In addition, all functions to be added to Larch need to be *registered*, by defining a function call `registerLarchPlugin()` that returns a tuple containing the name of the group containing the added functions, and a dictionary of Larch symbol names and functions. A simple plugin module would look like:

```
def _f1(x, y, larch=None):  # Note:  larch instance passed in with keyword
    if larch is None: return
    group = larch.symtable.create_group(name='created by f1')

    setattr(group, 'x', x) # add symbols by "setting attributes"
    setattr(group, 'y', y)

    return group

def registerLarchPlugin(): # must have a function with this name!
    return ('mymod', {'f1': _f1})
```

This is a fairly trivial example, simply putting data into a Group. Of course, the main point of a plugin is that you can do much more complicated work inside the function.

If this is placed in a file called 'myplugin.py' in the larch plugins folder (either $HOME/.larch/plugins/ or /usr/local/share/larch/plugins on Unix, or C:\Users\ME\larch\plugins or C:\Program Files\larch\plugins on Windows), then:

```
larch> add_plugin('myplugin')
```

will add a top-level group 'mymod' with an 'f1' function, so that:

```
larch> g1 = mymod.f1(10, 'yes')
larch> print g1
<Group created by f1!>
```

```
larch> print g1.x, g1.y
(10, 'yes')
```

For commonly used plugins, the `add_plugin()` call can be added to your startup script.

CHAPTER

FIVE

XAFS ANALYSIS WITH LARCH

One of the primary motivations for Larch was processing XAFS data.

CHAPTER

SIX

XRF ANALYSIS WITH LARCH

X-ray Fluorescence Data can be maniuplated and displayed with Larch.