Larch Documentation

Release 0.9.7

Matthew Newville

December 22, 2011

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 data are easy to manage and use.
- 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 in 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 Ifeffit verversio 2
- 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 but is not well-tested, and numpy, and scipy are ported to Python 3.X. The graphical system, based on wxWidgets has not yet been ported to Python 3.X.

In addition, numpy, matplotlib, and wxPython are required. These are simply installed as standard packages on almost all platforms.

All development is 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
```

notice, this list of conditions and the following disclaimer in the
documentation and/or other materials provided with the distribution.
* The names of the copyright holders or contributers may not be used to
endorse or promote products derived from this software without specific
prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS
IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

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/

2.1 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.

2.2 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. We'll get to this in a later section.

2.3 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 actually 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> sin(-1)
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.

2.4 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.

Here, the word "object" is used frequently. Each piece of data in Larch is a Python object, which is to say it has a value and may have specific functions that go with it.

2.4.1 Lists

A list is a sequence of other data types. The data types do not have to be the same type. 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]
```

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']
```

The syntax using a '.' indicates a method – a function specific to that object. 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:

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

but not:

```
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']
```

As always, consult the Python documentation for more details.

2.4.2 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. The syntax for tuples uses 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 they may seem less powerful than lists, tuples are actually used widely with Larch and Python, as once created a tuple has a predictable size and order to its elements. Thus, as with the example above, it can be used as a simple ordered container of data.

2.4.3 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...

2.4.4 Arrays

2.4.5 Dictionaries

2.5 Conditional Execution and Control-Flow

2.6 Object and Groups

2.6.1 Objects

2.7 Reading and Writing Data

2.8 Plotting and Displaying Data

2.9 Procedures

2.10 Dealing With Errors

CHAPTER

THREE

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.

3.1 Larch vs 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.

3.1.1 Code Blocks with 'end*'

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

```
if test:
   <block>
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.

3.1.2 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.

3.1.3 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

3.1.4 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

3.2 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.

3.3 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:UsersMElarchplugins or C:Program Fileslarchplugins 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

FOUR

LANGUAGE REFERENCE

This chapter describes further details of Larch language. The most important here is the fact that Larch is closely related to Python.

Of course, Python is well-documented and much of the Python documentation can be used for Larch. Thus the discussion here focusses on the differences with Python,

4.1 Language

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 well-tested, and

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.

CHAPTER

SEVEN

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*