Python 201 -- (Slightly) Advanced Python Topics

Python 201 -- (Slightly) Advanced Python Topics

Dave Kuhlman

http://www.rexx.com/~dkuhlman
Email: dkuhlman@rexx.com

Release 1.00
June 6, 2003

Front Matter

Abstract:

This document is a syllabus for a second course in Python programming. This course contains discussions of several advanced topics that are of interest to Python programmers.

Contents

1. Python 201 -- (Slightly) Advanced Python Topics

This document is intended as notes for a course on (slightly) advanced Python topics.

2. Regular Expressions

2.1 Defining regular expressions

Defining a regular expression is to provide a sequence of characters, the pattern, that will match sequences of characters in a target.

Here are several places to look for help:

- Python Library Reference: 4.2.1 Regular Expression Syntax
- Regular Expression HOWTO

The patterns or regular expressions can be defined as follows:

- Literal characters must match exactly. For example, "a" matches "a".
- Concatenated patterns match concatenated targets. For example, "ab" ("a" followed by "b") matches "ab".
- Alternate patterns, separated by a vertical bar, match either of the alternative patterns. For example, "(aaa)|(bbb)" will match either "aaa" or "bbb".
- Repeating and optional items:
    - "abc*" matches "ab" followed by *zero* or more occurances of "c", for example, "ab", "abc", "abcc", etc.
    - "abc+" matches "ab" followed by *one* or more occurances of "c", for example, "abc", "abcc", etc, but *not* "ab".
    - "abc?" matches "ab" followed by *zero or one* occurances of "c", for example, "ab" or "abc".
- Sets of characters -- Characters and sequences of characters in square brackets form a set; a set matches any character in the set or range. For example, "[abc]" matches "a" or "b" or "c". And, for example, "[_a-z0-9]" matches an underscore or any lower-case letter or any digit.
- Groups -- Parentheses indicate a group with a pattern. For example, "ab(cd)*ef" is a pattern that matches "ab" followed by any number of occurances of "cd" followed by "ef", for example, "abef", "abcdef", "abcdcdef", etc.

- There are special names for some sets of characters, for example "`\d`" (any digit), "`\w`" (any alphanumeric character), "`\W`" (any non-alphanumeric character), etc. See Python Library Reference: 4.2.1 Regular Expression Syntax for more.

Because of the use of backslashes in patterns, you are usually better off defining regular expressions with raw strings, e.g. `r"abc"`.

2.2 Compiling regular expressions

When a regular expression is to be used more than once, you should consider compiling it. For example:

```
import sys, re

pat = re.compile('aa[bc]*dd')

while 1:
    line = raw_input('Enter a line ("q" to quit):')
    if line == 'q':
        break
    if pat.search(line):
        print 'matched:', line
    else:
        print 'no match:', line
```

Comments:

- We import module `re` in order to use regular expresions.
- "`re.compile()`" compiles a regular expression so that we can reuse the compiled regular expression without compiling it repeatedly.

2.3 Using regular expressions

Use `match` to match at the beginning of a string (or not at all).

Use `search` to search a string and match the first string from the left.

Here are some examples:

```
>>> import re
>>> pat = re.compile('aa[0-9]*bb')
>>> x = pat.match('aa1234bbccddee')
>>> x
<_sre.SRE_Match object at 0x401e9608>
>>> x = pat.match('xxxxaa1234bbccddee')
>>> x
>>> type(x)
<type 'NoneType'>
```

```
>>> x = pat.search('xxxxaa1234bbccddee')
>>> x
<_sre.SRE_Match object at 0x401e9608>
```

Notes:

- When a match or search is successful, it returns a match object. When it fails, it returns *None*.
- You can also call the corresponding functions `match` and `search` in the `re` module, e.g.:

```
>>> x = re.search(pat, 'xxxxaa1234bbccddee')
>>> x
<_sre.SRE_Match object at 0x401e9560>
```

2.4 Using match objects to extract a value

Match objects enable you to extract matched sub-strings after performing a match. A match object is returned by successful match.

Here is an example:

```
import sys, re

pat = re.compile('aa([0-9]*)bb')

while 1:
    line = raw_input('Enter a line ("q" to quit):')
    if line == 'q':
        break
    mo = pat.search(line)
    if mo:
        value = mo.group(1)
        print 'value: %s' % value
    else:
        print 'no match'
```

Explanation:

- In the regular expression, put parentheses around the portion of the regular expression that will match what you want to extract. Each pair of parentheses marks off a group.
- After the search, check to determine if there was a successful match by checking for a matching object. "`pat.search(line)`" returns None if the search fails.
- If you specify more than one group in your regular expression (more that one pair of parentheses), then you can use "`value = mo.group(N)`" to extract the value matched by the Nth group from the matching object. "`value = mo.group(1)`"

returns the first extracted value; "`value = mo.group(2)`" returns the second; etc. An argument of 0 returns the string matched by the entire regular expression.

In addition, you can:

- Use "`values = mo.groups()`" to get a tuple containing the strings matched by all groups.
- Use "`mo.expand()`" to interpolate the group values into a string. For example, "`mo.expand(r'value1: \1 value2: \2')`"inserts the values of the first and second group into a string. If the first group matched "aaa" and the second matched "bbb", then this example would produce "value1: aaa value2: bbb".

2.5 Extracting multiple items

You can extract multiple items with a single search. Here is an example:

```
import sys, re

pat = re.compile('aa([0-9]*)bb([0-9]*)cc')

while 1:
    line = raw_input('Enter a line ("q" to quit):')
    if line == 'q':
        break
    mo = pat.search(line)
    if mo:
        value1, value2 = mo.group(1, 2)
        print 'value1: %s  value2: %s' % (value1, value2)
    else:
        print 'no match'
```

Comments:

- Use multiple parenthesized substrings in the regular expression to indicate the portions (groups) to be extracted.
- "`mo.group(1, 2)`" returns the values of the first and second group in the string matched.
- We could also have used "`mo.groups()`" to obtain a tuple that contains both values.
- Yet another alternative would have been to use the following; "`print mo.expand(r'value1: \1 value2: \2')`".

2.6 Replacing multiple items

You can locate sub-strings (slices) of a match and replace them. Here is an example:

```
import sys, re

pat = re.compile('aa([0-9]*)bb([0-9]*)cc')

while 1:
    line = raw_input('Enter a line ("q" to quit): ')
    if line == 'q':
        break
    mo = pat.search(line)
    if mo:
        value1, value2 = mo.group(1, 2)
        start1 = mo.start(1)
        end1 = mo.end(1)
        start2 = mo.start(2)
        end2 = mo.end(2)
        print 'value1: %s  start1: %d  end1: %d' % (value1, start1, end1)
        print 'value2: %s  start2: %d  end2: %d' % (value2, start2, end2)
        repl1 = raw_input('Enter replacement #1: ')
        repl2 = raw_input('Enter replacement #2: ')
        newline = line[:start1] + repl1 + line[end1:start2] + repl2 + line[
        print 'newline: %s' % newline
    else:
        print 'no match'
```

Download as text (original file name: Examples/python_201_re_repl.py).

Explanation:

- Alternatively, use "`mo.span(1)`" instead of "`mo.start(1)`" and "`mo.end(1)`" in order to get the start and end of a sub-match in a single operation. "`mo.span(1)`"returns a tuple: (start, end).
- Put together a new string with string concatenation from pieces of the original string and replacement values. You can use string slices to get the sub-strings of the original string. In our case, the following gets the start of the string, adds the first replacement, adds the middle of the original string, adds the second replacement, and finally, adds the last part of the original string:

```
newline = line[:start1] + repl1 + line[end1:start2] + repl2 + line[end
```

You can also use the `sub` function or method to do substitutions. Here is an example:

```
import sys, re

pat = re.compile('[0-9]+')

print 'Replacing decimal digits.'
while 1:
    target = raw_input('Enter a target line ("q" to quit): ')
    if target == 'q':
        break
```

```
    repl = raw_input('Enter a replacement: ')
    result = pat.sub(repl, target)
    print 'result: %s' % result
```

Download as text (original file name: Examples/python_201_re_repl2.py).

And, finally, you can define a function to be used to insert *calculated* replacements. Here is an example:

```
import sys, re, string

pat = re.compile('[a-m]+')

def replacer(mo):
    return string.upper(mo.group(0))

print 'Upper-casing a-m.'
while 1:
    target = raw_input('Enter a target line ("q" to quit): ')
    if target == 'q':
        break
    result = pat.sub(replacer, target)
    print 'result: %s' % result
```

Download as text (original file name: Examples/python_201_re_repl3.py).

Notes:

- If the replacement argument to `sub` is a function, that function must take one argument, a match object, and must return the modified (or replacement) value. The matched sub-string will be replaced by the value returned by this function.
- In our case, the function `replacer` converts the matched value to upper case.

This is also a convenient use for a `lambda` instead of a named function, for example:

```
import sys, re, string

pat = re.compile('[a-m]+')

print 'Upper-casing a-m.'
while 1:
    target = raw_input('Enter a target line ("q" to quit): ')
    if target == 'q':
        break
    result = pat.sub(
        lambda mo: string.upper(mo.group(0)),
        target)
    print 'result: %s' % result
```

Download as text (original file name: Examples/python_201_re_repl4.py).

3. Unit Tests

Unit test and the Python unit test framework provide a convenient way to define and run tests that ensure that a Python application produces specified results.

This section, while it will not attempt to explain everything about the unit test framework, will provide examples of several straight-forward ways to construct and run tests.

Some assumptions:

- We are going to develop a software project incrementally. We will *not* implement and release all at once. Therefore, each time we add to our existing code base, we need a way to verify that our additions (and fixes) have not caused new problems in old code.
- Adding new code to existing code *will* cause problems.

3.1 Defining unit tests

1. Create a test class.
2. In the test class, implement a number of methods to perform your tests. Name your test methods with the prefix "test". Here is an example:

```
class MyTest:
    def test_one(self):
        # some test code
        pass
    def test_two(self):
        # some test code
        pass
```

3. Create a test harness. Here is an example:

```
# make the test suite.
def suite():
    loader = unittest.TestLoader()
    testsuite = loader.loadTestsFromTestCase(MyTest)
    return testsuite

# Make the test suite; run the tests.
def test():
    testsuite = suite()
    runner = unittest.TextTestRunner(sys.stdout, verbosity=2)
    result = runner.run(testsuite)
```

Here is a more complete example:

```
import sys, StringIO, string
import unittest
import webserv_example_heavy_sub

# A comparison function for case-insenstive sorting.
def mycmpfunc(arg1, arg2):
    return cmp(string.lower(arg1), string.lower(arg2))

class XmlTest(unittest.TestCase):
    def test_import_export1(self):
        inFile = file('test1_in.xml', 'r')
        inContent = inFile.read()
        inFile.close()
        doc = webserv_example_heavy_sub.parseString(inContent)
        outFile = StringIO.StringIO()
        outFile.write('<?xml version="1.0" ?>\n')
        doc.export(outFile, 0)
        outContent = outFile.getvalue()
        outFile.close()
        self.failUnless(inContent == outContent)

# make the test suite.
def suite():
    loader = unittest.TestLoader()
    # Change the test method prefix: test --> trial.
    #loader.testMethodPrefix = 'trial'
    # Change the comparison function that determines the order of tests.
    #loader.sortTestMethodsUsing = mycmpfunc
    testsuite = loader.loadTestsFromTestCase(XmlTest)
    return testsuite

# Make the test suite; run the tests.
def test_main():
    testsuite = suite()
    runner = unittest.TextTestRunner(sys.stdout, verbosity=2)
    result = runner.run(testsuite)

if __name__ == "__main__":
    test_main()
```

Download as text (original file name: Examples/python_201_utest_1.py).

Running the above script produces the following output:

```
test_import_export (__main__.XmlTest) ... ok

----------------------------------------------------------------------
Ran 1 test in 0.035s

OK
```

A few notes on this example:

- This example tests the ability to parse an xml document test1_in.xml and export that document back to XML. The test succeeds if the input XML document and the exported XML document are the same.
- The code which is being tested parses an XML document returned by a request to Amazon Web services. You can learn more about Amazon Web services at: http://www.amazon.com/webservices. This code was generated from an XML Schema document by **generateDS.py**. So we are in effect, testing **generateDS.py**. You can find **generateDS.py** at: http://www.rexx.com/~dkuhlman/#generateDS.
- Testing for success/failure and reporting failures -- Use the methods listed at http://www.python.org/doc/current/lib/testcase-objects.html to test for and report success and failure. In our example, we used "`self.failUnless(inContent == outContent)`" to ensure that the content we parsed and the content that we exported were the same.
- Add additional tests by adding methods whose names have the prefix "test". If you prefer a different prefix for tests names, add something like the following to the above script:

  ```
  loader.testMethodPrefix = 'trial'
  ```

- By default, the tests are run in the order of their names sorted by the `cmp` function. So, if need to, you can control the order of execution of tests by selecting their names, for example, using names like test_1_checkderef, test_2_checkcalc, etc. Or, you can change the comparison function by adding something like the following to the above script:

  ```
  loader.sortTestMethodsUsing = mycmpfunc
  ```

As a bit of motivation for creating and using unit tests, while developing this example, I discovered several errors (or maybe "special features") in **generateDS.py**.

4. Extending and embedding Python

4.1 Introduction and concepts

Extending vs. embedding -- They are different but related:

- Extending Python means to implement an extension module or an extension type. An extension *module* creates a new Python module which is implemented in C/C++. From Python code, an extension module appears to be just like a module implemented in Python code. An extension *type* creates a new Python (built-in) type which is implemented in C/C++. From Python code, an extension type appears to be just like a built-in type.

- Embedding Python, by contrast, is to put the Python interpreter within an application (i.e. link it in) so that the application can run Python scripts. The scripts can be executed or triggered in a variety of ways, e.g. they can be bound to keys on the keyboard or to menu items, they can be triggered by external events, etc. Usually, in order to make the embedded Python interpreter useful, Python is also extended with functions from the embedding application, so that the scripts can call functions that are implemented by the embedding C/C++ application.

Documentation -- The two important sources for information about extending and embedding are the following:

- Extending and Embedding the Python Interpreter
- Python/C API Reference Manual

Types of extensions:

- Extension modules -- From the Python side, it appears to be a Python module. Usually it exports functions.
- Extension types -- Used to implement a new Python data type.
- Extension classes -- From the Python side, it appears to be a class.

Tools -- There are several tools that support the development of Python extensions:

- SWIG -- Learn about SWIG at: http://www.swig.org.
- Pyrex -- Learn about Pyrex at: http://www.cosc.canterbury.ac.nz/~greg/python/Pyrex/.

4.2 Extension modules

Writing an extension module by hand -- What to do:

- Create the "init" function -- The name of this function must be "init" followed by the name of the module. Every extension module must have such a function.
- Create the function table -- This table maps function names (referenced from Python code) to function pointers (implemented in C/C++).
- Implement each wrapper function.

Implementing a wrapper function -- What to do:

1. Capture the arguments with `PyArg_ParseTuple`. The format string specifies how arguments are to be converted and captured. See 1.7 Extracting Parameters in Extension Functions. Here are some of the most commonly used types:

- Use "i", "s", "f", etc to convert and capture simple types such as integers, strings, floats, etc.
- Use "O" to get a pointer to Python "complex" types such as lists, tuples, dictionaries, etc.
- Use items in parentheses to capture and unpack sequences (e.g. lists and tuples) of fixed length. Example:

```
if (!PyArg_ParseTuple(args, "(ii)(ii)", &x, &y, &width, &height))
{
    return NULL;
} /* if */
```

  A sample call might be:

```
lowerLeft = (x1, y1)
extent = (width1, height1)
scan(lowerLeft, extent)
```

- Use ":aName" (colon) at the end of the format string to provide a function name for error messages. Example:

```
if (!PyArg_ParseTuple(args, "O:setContentHandler", &pythonInstance
{
    return NULL;
} /* if */
```

- Use ";an error message" (semicolon) at the end of the format string to provide a string that replaces the default error message.
- Docs are available at: http://www.python.org/doc/current/ext/parseTuple.html.

2. Write the logic.
3. Handle errors and exceptions -- You will need to understand how to (1) clearing errors and exceptions and (2) Raise errors (exceptions).

   - Many functions in the Python C API raise exceptions. You will need to check for and clear these exceptions. Here is an example:

```
char * message;
int messageNo;

message = NULL;
messageNo = -1;
/* Is the argument a string?
*/
if (! PyArg_ParseTuple(args, "s", &message))
{
    /* It's not a string.  Clear the error.
```

```
    *  Then try to get a message number (an integer).
    */
    PyErr_Clear();
    if (! PyArg_ParseTuple(args, "i", &messageNo))
    {
        o
        o
        o
```

- You can also raise exceptions in your C code that can be caught (in a "`try:except:`" block) back in the calling Python code. Here is an example:

  ```
  if (n == 0)
  {
      PyErr_SetString(PyExc_ValueError, "Value must not be zero");
      return NULL;
  }
  ```

  See Include/pyerrors.h in the Python source distribution for more exception/error types.

- And, you can test whether a function in the Python C API that you have called has raised an exception. For example:

  ```
  if (PyErr_Occurred())
  {
      /* An exception was raised.
      *  Do something about it.
      */
      o
      o
      o
  ```

For more documentation on errors and exceptions, see: http://www.python.org/doc/current/api/exceptionHandling.html.

4. Create and return a value:

   - For each built-in Python type there is a set of API functions to create and manipulate it. See the "Python/C API Reference Manual" for a description of these functions. For example, see:

     - http://www.python.org/doc/current/api/intObjects.html
     - http://www.python.org/doc/current/api/stringObjects.html
     - http://www.python.org/doc/current/api/tupleObjects.html
     - http://www.python.org/doc/current/api/listObjects.html
     - http://www.python.org/doc/current/api/dictObjects.html

        - Etc.

    - The reference count -- You will need to follow Python's rules for reference counting that Python uses to garbage collect objects. You can learn about these rules at http://www.python.org/doc/current/ext/refcounts.html. You will not want Python to garbage collect objects that you create too early or too late. With respect to Python objects created with the above functions, these new objects are owned and may be passed back to Python code. However, there are situations where your C/C++ code will not automatically own a reference, for example when you extract an object from a container (a list, tuple, dictionary, etc). In these cases you should increment the reference count with *Py_INCREF*.

4.3 SWIG

Note: Our discussion and examples are for SWIG version 1.3

SWIG will often enable you to generate wrappers for functions in an existing C function library. SWIG does not understand everything in C header files. But it does a fairly impressive job. You should try it first before resorting to the hard work of writing wrappers by hand.

More information on **SWIG** is at http://www.swig.org.

Here are some steps that you can follow:

1. Create an interface file -- Even when you are wrapping functions defined in an existing header file, creating an interface file is a good idea. Include your existing header file into it, then add whatever else you need. Here is an extremely simple example of a SWIG interface file:

    ```
    %module MyLibrary

    %{
    #include "MyLibrary.h"
    %}

    %include "MyLibrary.h"
    ```

    Download as text (original file name: Examples/MyLibrary.i).

    Comments:

    - The "%{" and "%}" brackets are directives to SWIG. They say: "Add the code between these brackets to the *generated* wrapper file without processing it.

    - The "%include" statement says: "Copy the file into the *interface* file here. In effect, you are asking SWIG to generate wrappers for *all* the functions in this header file. If you want wrappers for only *some* of the functions in a header

file, then copy or reproduce function declarations for the desired functions here. An example:

```
%module MyLibrary

%{
#include "MyLibrary.h"
%}

int calcArea(int width, int height);
int calcVolume(int radius);
```

This example will generate wrappers for only *two* functions.

- You can find more information about the directives that are used in SWIG interface files in the SWIG User Manual, in particular at:
  - http://www.swig.org/Doc1.3/Preprocessor.html
  - http://www.swig.org/Doc1.3/Python.html

2. Generate the wrappers:

```
swig -python MyLibrary.i
```

3. Compile and link the library. On Linux, you can use something like the following:

```
gcc -c MyLibrary.c
gcc -c -I/usr/local/include/python2.3 MyLibrary_wrap.c
gcc -shared MyLibrary.o MyLibrary_wrap.o -o _MyLibrary.so
```

Note that we produce a shared library whose name is the module name prefixed with an underscore. **SWIG** also generates a .py file, without the leading underscore, which we will import from our Python code and which, in turn, imports the shared library.

4. Use the extension module in your python code:

```
Python 2.3b1 (#1, Apr 25 2003, 20:36:09)
[GCC 2.95.4 20011002 (Debian prerelease)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import MyLibrary
>>> MyLibrary.calcArea(4.0, 5.0)
20.0
```

Here is a makefile that will execute swig to generate wrappers, then compile and link the extension.

```
CFLAGS = -I/usr/local/include/python2.3

all: _MyLibrary.so
```

```
_MyLibrary.so: MyLibrary.o MyLibrary_wrap.o
        gcc -shared MyLibrary.o MyLibrary_wrap.o -o _MyLibrary.so

MyLibrary.o: MyLibrary.c
        gcc -c MyLibrary.c -o MyLibrary.o

MyLibrary_wrap.o: MyLibrary_wrap.c
        gcc -c ${CFLAGS} MyLibrary_wrap.c -o MyLibrary_wrap.o

MyLibrary_wrap.c: MyLibrary.i
        swig -python MyLibrary.i

clean:
        rm -f MyLibrary.py MyLibrary.o MyLibrary_wrap.c \
                MyLibrary_wrap.o _MyLibrary.so
```

Download as text (original file name: Examples/MyLibrary_makefile).

Here is an example of running this makefile:

```
$ make -f MyLibrary_makefile clean
rm -f MyLibrary.py MyLibrary.o MyLibrary_wrap.c \
        MyLibrary_wrap.o _MyLibrary.so
$ make -f MyLibrary_makefile
gcc -c MyLibrary.c -o MyLibrary.o
swig -python MyLibrary.i
gcc -c -I/usr/local/include/python2.3 MyLibrary_wrap.c -o MyLibrary_wrap.o
gcc -shared MyLibrary.o MyLibrary_wrap.o -o _MyLibrary.so
```

And, here are C source files that can be used in our example:

```
/* MyLibrary.h
*/

float calcArea(float width, float height);
float calcVolume(float radius);

int getVersion();

int getMode();
```

Download as text (original file name: Examples/MyLibrary.h).

```
/* MyLibrary.c
*/

float calcArea(float width, float height)
{
   return (width * height);
}
```

```
float calcVolume(float radius)
{
   return (3.14 * radius * radius);
}

int getVersion()
{
        return 123;
}

int getMode()
{
        return 1;
}
```

Download as text (original file name: Examples/MyLibrary.c).

4.4 Pyrex

Pyrex is a useful tool for writing Python extensions. Because the **Pyrex** language is similar to Python, writing extensions in **Pyrex** is easier than doing so in C.

More information on **Pyrex** is at http://www.cosc.canterbury.ac.nz/~greg/python/Pyrex/.

Here is a simple function definition in **Pyrex**.

```
# python_201_pyrex_string.pyx

import string

def formatString(object s1, object s2):
    s1 = string.strip(s1)
    s2 = string.strip(s2)
    s3 = '<<%s||%s>>' % (s1, s2)
    s4 = s3 * 4
    return s4
```

Download as text (original file name: Examples/python_201_pyrex_string.pyx).

And, here is a make file:

```
CFLAGS = -DNDEBUG -O3 -Wall -Wstrict-prototypes -fPIC \
        -I/usr/local/include/python2.3

all: python_201_pyrex_string.so
```

```
python_201_pyrex_string.so: python_201_pyrex_string.o
        gcc -shared python_201_pyrex_string.o -o python_201_pyrex_string.sc

python_201_pyrex_string.o: python_201_pyrex_string.c
        gcc -c ${CFLAGS} python_201_pyrex_string.c -o python_201_pyrex_stri

python_201_pyrex_string.c: python_201_pyrex_string.pyx
        pyrexc python_201_pyrex_string.pyx

clean:
        rm -f python_201_pyrex_string.so python_201_pyrex_string.o \
                python_201_pyrex_string.c
```

Download as text (original file name: Examples/python_201_pyrex_makestring).

Here is another example. In this one, one function in the .pyx file calls another. Here is the implementation file:

```
# python_201_pyrex_primes.pyx

def showPrimes(int kmax):
    plist = primes(kmax)
    for p in plist:
        print 'prime: %d' % p

cdef primes(int kmax):
    cdef int n, k, i
    cdef int p[1000]
    result = []
    if kmax > 1000:
        kmax = 1000
    k = 0
    n = 2
    while k < kmax:
        i = 0
        while i < k and n % p[i] <> 0:
            i = i + 1
        if i == k:
           p[k] = n
           k = k + 1
           result.append(n)
        n = n + 1
    return result
```

Download as text (original file name: Examples/python_201_pyrex_primes.pyx).

And, here is a make file:

```
#CFLAGS = -DNDEBUG -g -O3 -Wall -Wstrict-prototypes -fPIC \
#       -I/usr/local/include/python2.3
```

```
CFLAGS = -DNDEBUG -I/usr/local/include/python2.3

all: python_201_pyrex_primes.so

python_201_pyrex_primes.so: python_201_pyrex_primes.o
        gcc -shared python_201_pyrex_primes.o -o python_201_pyrex_primes.so

python_201_pyrex_primes.o: python_201_pyrex_primes.c
        gcc -c ${CFLAGS} python_201_pyrex_primes.c -o python_201_pyrex_prim

python_201_pyrex_primes.c: python_201_pyrex_primes.pyx
        pyrexc python_201_pyrex_primes.pyx

clean:
        rm -f python_201_pyrex_primes.so python_201_pyrex_primes.o \
                python_201_pyrex_primes.c
```

Download as text (original file name: Examples/python_201_pyrex_makeprimes).

Here is the output from running the makefile:

```
$ make -f python_201_pyrex_makeprimes clean
rm -f python_201_pyrex_primes.so python_201_pyrex_primes.o \
        python_201_pyrex_primes.c
$ make -f python_201_pyrex_makeprimes
pyrexc python_201_pyrex_primes.pyx
gcc -c -DNDEBUG -I/usr/local/include/python2.3  python_201_pyrex_primes.c -
gcc -shared python_201_pyrex_primes.o -o python_201_pyrex_primes.so
```

Here is an interactive example of its use:

```
$ python
Python 2.3b1 (#1, Apr 25 2003, 20:36:09)
[GCC 2.95.4 20011002 (Debian prerelease)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import python_201_pyrex_primes
>>> dir(python_201_pyrex_primes)
['__builtins__', '__doc__', '__file__', '__name__', 'showPrimes']
>>> python_201_pyrex_primes.showPrimes(5)
prime: 2
prime: 3
prime: 5
prime: 7
prime: 11
```

This next example shows how to use Pyrex to implement a new extension *type*, that is a new Python built-in type. Notice that the class is declared with the `cdef` keyword, which tells Pyrex to generate the C implementation of a type instead of a class.

Here is the implementation file:

```
# python_201_pyrex_clsprimes.pyx

"""An implementation of primes handling class
for a demonstration of Pyrex.
"""

cdef class Primes:
    """A class containing functions for
    handling primes.
    """

    def showPrimes(self, int kmax):
        """Show a range of primes.
        Use the method primes() to generate the primes.
        """
        plist = self.primes(kmax)
        for p in plist:
            print 'prime: %d' % p

    def primes(self, int kmax):
        """Generate the primes in the range 0 - kmax.
        """
        cdef int n, k, i
        cdef int p[1000]
        result = []
        if kmax > 1000:
            kmax = 1000
        k = 0
        n = 2
        while k < kmax:
            i = 0
            while i < k and n % p[i] <> 0:
                i = i + 1
            if i == k:
               p[k] = n
               k = k + 1
               result.append(n)
            n = n + 1
        return result
```

Download as text (original file name: Examples/python_201_pyrex_clsprimes.pyx).

And, here is a make file:

```
CFLAGS = -DNDEBUG -I/usr/local/include/python2.3

all: python_201_pyrex_clsprimes.so

python_201_pyrex_clsprimes.so: python_201_pyrex_clsprimes.o
        gcc -shared python_201_pyrex_clsprimes.o -o python_201_pyrex_clspri
```

```
python_201_pyrex_clsprimes.o: python_201_pyrex_clsprimes.c
        gcc -c ${CFLAGS} python_201_pyrex_clsprimes.c -o python_201_pyrex_c

python_201_pyrex_clsprimes.c: python_201_pyrex_clsprimes.pyx
        pyrexc python_201_pyrex_clsprimes.pyx

clean:
        rm -f python_201_pyrex_clsprimes.so python_201_pyrex_clsprimes.o \
                python_201_pyrex_clsprimes.c
```

Download as text (original file name: Examples/python_201_pyrex_makeclsprimes).

Here is output from running the makefile:

```
$ make -f python_201_pyrex_makeclsprimes clean
rm -f python_201_pyrex_clsprimes.so python_201_pyrex_clsprimes.o \
        python_201_pyrex_clsprimes.c
$ make -f python_201_pyrex_makeclsprimes
pyrexc python_201_pyrex_clsprimes.pyx
gcc -c -DNDEBUG -I/usr/local/include/python2.3  python_201_pyrex_clsprimes.
gcc -shared python_201_pyrex_clsprimes.o -o python_201_pyrex_clsprimes.so
```

And here is an interactive example of its use:

```
$ python
Python 2.3b1 (#1, Apr 25 2003, 20:36:09)
[GCC 2.95.4 20011002 (Debian prerelease)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import python_201_pyrex_clsprimes
>>> dir(python_201_pyrex_clsprimes)
['Primes', '__builtins__', '__doc__', '__file__', '__name__']
>>> primes = python_201_pyrex_clsprimes.Primes()
>>> dir(primes)
['__class__', '__delattr__', '__doc__', '__getattribute__', '__hash__',
'__init__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__str__', 'primes', 'showPrimes']
>>> primes.showPrimes(4)
prime: 2
prime: 3
prime: 5
prime: 7
```

Documentation -- Also notice that **Pyrex** preserves the documentation for the module, the class, and the methods in the class. You can show this documentation with **pydoc**, as follows:

```
$ pydoc python_201_pyrex_clsprimes
```

Or, in Python interactive mode, use:

```
$ python
```

```
Python 2.3b1 (#1, Apr 25 2003, 20:36:09)
[GCC 2.95.4 20011002 (Debian prerelease)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import python_201_pyrex_clsprimes
>>> help(python_201_pyrex_clsprimes)
```

4.5 SWIG vs. Pyrex

Choose **SWIG** when:

- You already have an existing C or C++ implementation of the code you want to call from Python. In this case you want **SWIG** to generate the wrappers.
- You want to write the implementation in C or C++ by hand. Perhaps, because you think you can do so quickly, for example, or because you believe that you can make it highly optimized. Then, you want to be able to generate the Python (extension) wrappers for it quickly.

Choose **Pyrex** when:

- You do *not* have a C/C++ implementation and you want an easier way to write that C implementation. Writing **Pyrex** code, which is a lot like Python, is easier than writing C or C++ code by hand).
- You start to write the implementation in C, then find that it requires lots of calls to the Python C API, and you want to avoid having to learn how to do that.

4.6 Extension types

The goal -- A new built-in data type for Python.

Existing examples -- Objects/listobject.c, Objects/stringobject.c, Objects/dictobject.c, etc in the Python source code distribution.

In older versions of the Python source code distribution, a template for the C code was provided in Objects/xxobject.c. Objects/xxobject.c is no longer included in the Python source code distribution. However:

- The discussion and examples for creating extension types have been expanded. See: *Extending and Embedding the Python Interpreter*, 2. Defining New Types.
- In the Tools/framer directory of the Python source code distribution there is an application that will generate a skeleton for an extension type from a specification object written in Python. Run Tools/framer/example.py to see it in action.

And, you can use Pyrex to generate a new built-in type. To do so, implement a Python/Pyrex class and declare the class with the **Pyrex** keyword `cdef`. In fact, you may want to use Pyrex to generate a minimal extension type, and then edit that generated code to insert and add functionality by hand. See the **Pyrex** section for an

example.

Pyrex also goes some way toward giving you access to (existing) C structs and functions from Python.

4.7 Extension classes

Extension classes the easy way -- SWIG shadow classes.

Start with an implementation of a C++ class and its header file.

Use the following SWIG flags:

```
swig -c++ -python mymodule.i
```

More information is available with the SWIG documentation at: http://www.swig.org/Doc1.3/Python.html.

Extension classes the **Pyrex** way -- An alternatie is to use **Pyrex** to compile a class definition that does *not* have the `cdef` keyword. Using `cdef` on the class tells Pyrex to generate an extension *type* instead of a class. You will have to determine whether you want an extension class or an extension type.

5. Parsing

Python is an excellent language for text analysis.

In some cases, simply splitting lines of text into words will be enough. In these cases use `string.split()`.

In other cases, regular expressions may be able to do the parsing you need. If so, see the section on regular expressions in this document.

However, in some cases, more complex analysis of input text is required. This section describes some of the ways that Python can help you with this complex parsing and analysis.

5.1 Special purpose parsers

There are a number of special purpose parsers which you will find in the Python standard library:

- ConfigParser -- Configuration file parser. See Python Library Reference: 5.10 ConfigParser - Configuration file parser.
- getopt -- Parse command line arguments. See Python Library Reference: 6.18 getopt - Parser for command line options.

- optparse -- Powerful parser for command line arguments. (Added in Python 2.3.)
- urlparse -- Parse URLs into components. See Python Library Reference: 11.14 urlparse - Parse URLs into components.
- `os.path` -- Parsers (and other capabilities) for file paths and names. See Python Library Reference: 6.2 os.path - Common pathname manipulations.
- PyXML and the XML parsers -- There is lots of support for parsing and processing XML. Here are a few places to look for support:
  - The Python standard library -- Python Library Reference: 13. Structured Markup Processing Tools.
  - PyXML -- http://www.python.org/sigs/xml-sig/.
  - Gnosis -- http://www.gnosis.cx/download/.
  - libxml2 and libxslt -- http://xmlsoft.org.
  - Dave' support for Python and XML -- http://www.rexx.com/~dkuhlman.

5.2 Writing a recursive descent parser by hand

For simple grammars, this is not so hard.

You will need to implement:

- A recognizer method or function for each production rule in your grammar. Each recognizer method begins looking at the current token, then consumes as many tokens as needed to recognize it's own production rule. It calls the recognizer functions for any non-terminals on its right-hand side.
- A tokenizer -- Something that will enable each recognizer function to get tokens, one by one. There are a variety of ways to do this, e.g. (1) a function that produces a list of tokens from which recognizers can pop tokens; (2) a generator whose `next` method returns the next token; etc.

Here is an example of a recursive descent parser written in Python. After the example is some explanation.

```
#!/usr/bin/env python

"""
python_201_rparser.py

A recursive descent parser example.
```

```
The grammar:

    Prog ::= Command | Command Prog
    Command ::= Func_call
    Func_call ::= Term '(' Func_call_list ')'
    Func_call_list ::= Func_call | Func_call ',' Func_call_list
    Term = <word>
"""

import sys, string, types
import getopt

## from IPython.Shell import IPShellEmbed
## ipshell = IPShellEmbed((),
##     banner = '>>>>>>>> Into IPython >>>>>>>>',
##     exit_msg = '<<<<<<<< Out of IPython <<<<<<<<')

#
# Constants
#

# AST node types
NoneNodeType = 0
ProgNodeType = 1
CommandNodeType = 2
FuncCallNodeType = 3
FuncCallListNodeType = 4
TermNodeType = 5

# Token types
NoneTokType = 0
LParTokType = 1
RParTokType = 2
WordTokType = 3
CommaTokType = 4
EOFTokType = 5

# Dictionary to map node type values to node type names
NodeTypeDict = {
    NoneNodeType: 'NoneNodeType',
    ProgNodeType: 'ProgNodeType',
    CommandNodeType: 'CommandNodeType',
    FuncCallNodeType: 'FuncCallNodeType',
    FuncCallListNodeType: 'FuncCallListNodeType',
    TermNodeType: 'TermNodeType',
    }

#
# Representation of a node in the AST (abstract syntax tree).
#
class ASTNode:
```

```
        def __init__(self, nodeType, *args):
            self.nodeType = nodeType
            self.children = []
            for item in args:
                self.children.append(item)
        def show(self, level):
            self.showLevel(level)
            print 'Node -- Type %s' % NodeTypeDict[self.nodeType]
            level += 1
            for child in self.children:
                if isinstance(child, ASTNode):
                    child.show(level)
                elif type(child) == types.ListType:
                    for item in child:
                        item.show(level)
                else:
                    self.showLevel(level)
                    print 'Child:', child
        def showLevel(self, level):
            for idx in range(level):
                print '   ',

    #
    # The recursive descent parser class.
    #   Contains the "recognizer" methods, which implement the grammar
    #   rules (above), one recognizer method for each production rule.
    #
    class ProgParser:
        def __init__(self):
            pass

        def parseFile(self, infileName):
            self.infileName = infileName
            self.tokens = None
            self.tokenType = NoneTokType
            self.token = ''
            self.lineNo = -1
            self.infile = file(self.infileName, 'r')
            self.tokens = genTokens(self.infile)
            try:
                self.tokenType, self.token, self.lineNo = self.tokens.next()
            except StopIteration:
                raise RuntimeError, 'Empty file'
            result = self.prog_reco()
            self.infile.close()
            self.infile = None
            return result

        def parseStream(self, instream):
            self.tokens = genTokens(instream, '<instream>')
            try:
```

```
            self.tokenType, self.token, self.lineNo = self.tokens.next()
        except StopIteration:
            raise RuntimeError, 'Empty file'
        result = self.prog_reco()
        return result

    def prog_reco(self):
        commandList = []
        while 1:
            result = self.command_reco()
            if not result:
                break
            commandList.append(result)
        return ASTNode(ProgNodeType, commandList)

    def command_reco(self):
        if self.tokenType == EOFTokType:
            return None
        result = self.func_call_reco()
        return ASTNode(CommandNodeType, result)

    def func_call_reco(self):
        if self.tokenType == WordTokType:
            term = ASTNode(TermNodeType, self.token)
            self.tokenType, self.token, self.lineNo = self.tokens.next()
            if self.tokenType == LParTokType:
                self.tokenType, self.token, self.lineNo = self.tokens.next(
                result = self.func_call_list_reco()
                if result:
                    if self.tokenType == RParTokType:
                        self.tokenType, self.token, self.lineNo = \
                            self.tokens.next()
                        return ASTNode(FuncCallNodeType, term, result)
                    else:
                        raise ParseError(self.lineNo, 'missing right paren'
                else:
                    raise ParseError(self.lineNo, 'bad func call list')
            else:
                raise ParseError(self.lineNo, 'missing left paren')
        else:
            return None

    def func_call_list_reco(self):
        terms = []
        while 1:
            result = self.func_call_reco()
            if not result:
                break
            terms.append(result)
            if self.tokenType != CommaTokType:
                break
```

```
            self.tokenType, self.token, self.lineNo = self.tokens.next()
        return ASTNode(FuncCallListNodeType, terms)

#
# The parse error exception class.
#
class ParseError(Exception):
    def __init__(self, lineNo, msg):
        RuntimeError.__init__(self, msg)
        self.lineNo = lineNo
        self.msg = msg
    def getLineNo(self):
        return self.lineNo
    def getMsg(self):
        return self.msg

def is_word(token):
    for letter in token:
        if letter not in string.ascii_letters:
            return None
    return 1

#
# Generate the tokens.
# Usage:
#    gen = genTokens(infile)
#    tokType, tok, lineNo = gen.next()
#    ...
def genTokens(infile):
    lineNo = 0
    while 1:
        lineNo += 1
        try:
            line = infile.next()
        except:
            yield (EOFTokType, None, lineNo)
        toks = line.split()
        for tok in toks:
            if is_word(tok):
                tokType = WordTokType
            elif tok == '(':
                tokType = LParTokType
            elif tok == ')':
                tokType = RParTokType
            elif tok == ',':
                tokType = CommaTokType
            yield (tokType, tok, lineNo)

def test(infileName):
    parser = ProgParser()
    #ipshell('(test) #1\nCtrl-D to exit')
```

```
        result = None
        try:
            result = parser.parseFile(infileName)
        except ParseError, exp:
            sys.stderr.write('ParseError: (%d) %s\n' % \
                (exp.getLineNo(), exp.getMsg()))
        if result:
            result.show(0)

    USAGE_TEXT = """
    Usage:
        python rparser.py [options] <inputfile>
    Options:
        -h, --help      Display this help message.
    Example:
        python rparser.py myfile.txt
    """

    def usage():
        print USAGE_TEXT
        sys.exit(-1)

    def main():
        args = sys.argv[1:]
        try:
            opts, args = getopt.getopt(args, 'h', ['help'])
        except:
            usage()
        relink = 1
        for opt, val in opts:
            if opt in ('-h', '--help'):
                usage()
        if len(args) != 1:
            usage()
        test(args[0])

    if __name__ == '__main__':
        main()
        #import pdb
        #pdb.run('main()')
```

Download as text (original file name: Examples/python_201_rparser.py).

And, here is a sample of the data we can apply this parser to:

```
aaa ( )
bbb ( ccc ( ) )
ddd ( eee ( ) , fff ( ggg ( ) , hhh ( ) , iii ( ) ) )
```

Download as text (original file name: Examples/python_201_rparser_data.txt).

Comments and explanation:

- The tokenizer is a Python generator. It returns a Python generator that can produce "`(tokType, tok, lineNo)`" tuples. Our tokenizer is so simple-minded that we have to separate all of our tokens with whitespace. (A little later, we'll see how to use **Plex** to overcome this limitation.)
- The parser class (`ProgParser`) contains the recognizer methods that implement the production rules. Each of these methods *recognizes* a syntactic construct defined by a rule. In our example, these methods have names that end with "_reco".
- We could have, alternatively, implemented our recognizers as global functions, instead of as methods in a class. However, using a class gives us a place to "hang" the variables that are needed across methods and saves us from having to use ("evil") global variables.
- A recognizer method recognizes a terminals (syntactic elements on the right-hand side of the grammar rule for which there *is no* grammar rule) by (1) checking the token type and the token value, and then (2) calling the tokenizer to get the next token (because it has consumed a token).
- A recognizer method checks for and processes a *non-terminal* (syntactic elements on the right-hand side for which there *is* a grammar rule) by calling the recognizer method that implements that non-terminal.
- If a recognizer method finds a syntax error, it raises an exception of class `ParserError`.
- Since our example recursive descent parser creates an AST (an abstract syntax tree), whenever a recognizer method successfully recognizes a syntactic construct, it creates an instance of class `ASTNode` to represent it and returns that instance to its caller. The instance of `ASTNode` has a node type and contains child nodes which were constructed by recognizer methods called by this one (i.e. that represent non-terminals on the right-hand side of a grammar rule).
- Each time a recognizer method "consumes a token", it calls the tokenizer to get the next token (and token type and line number).
- The tokenizer returns a token type in addition to the token value. It also returns a line number for error reporting.
- The syntax tree is constructed from instances of class `ASTNode`.
- The `ASTNode` class has a `show` method, which walks the AST and produces output. You can imagine that a similar method could do code generation. And, you should consider the possibility of writing analogous tree walk methods that perform tasks such as optimization, annotation of the AST, etc.

5.3 Creating a lexer/tokenizer with Plex

Lexical analysis -- The tokenizer in our recursive descent parser example was (for demonstration purposes) overly simple. You can always write more complex tokenizers by hand. However, for more complex (and real) tokenizers, you may want to use a tool to build your tokenizer.

In this section we'll describe **Plex** and use it to produce a tokenizer for our recursive descent parser.

You can obtain **Plex** at http://www.cosc.canterbury.ac.nz/~greg/python/Plex/.

In order to use it, you may want to add Plex-1.1.4/Plex to your PYTHONPATH.

Here is a simple example from the Plex tutorial:

```
#!/usr/bin/env python

# python_201_plex1.py
#
# Sample Plex lexer
#

import sys
import Plex

def test(infileName):
    letter = Plex.Range("AZaz")
    digit =  Plex.Range("09")
    name = letter +  Plex.Rep(letter | digit)
    number =  Plex.Rep1(digit)
    space =  Plex.Any(" \t\n")
    comment =  Plex.Str('"') +  Plex.Rep( Plex.AnyBut('"')) +  Plex.Str('"
    resword =  Plex.Str("if", "then", "else", "end")
    lexicon =  Plex.Lexicon([
        (resword,         'keyword'),
        (name,            'ident'),
        (number,          'int'),
        ( Plex.Any("+-*/=<>"),   Plex.TEXT),
        (space,  Plex.IGNORE),
        (comment, 'comment'),
    ])
    infile = open(infileName, "r")
    scanner =  Plex.Scanner(lexicon, infile, infileName)
    while 1:
        token = scanner.read()
        position = scanner.position()
        print '(%d, %d) tok: %s  tokType: %s' % \
            (position[1], position[2], token[1], token[0])
        if token[0] is None:
```

```
            break

USAGE_TEXT = """
Usage: python python_201_plex1.py <infile>
"""

def usage():
    print USAGE_TEXT
    sys.exit(-1)

def main():
    args = sys.argv[1:]
    if len(args) != 1:
        usage()
    infileName = args[0]
    test(infileName)

if __name__ == '__main__':
    main()
    #import pdb
    #pdb.run('main()')
```

Download as text (original file name: Examples/python_201_plex1.py).

Comments and explanation:

- Create a *lexicon* from scanning patterns.
- See the Plex tutorial and reference (and below) for more information on how to construct the patterns that match various tokens.
- Create a scanner with a *lexicon*, an input file, and an input file name.
- The call "`scanner.read()`" gets the next token. It returns a tuple containing (1) the token value and (2) the token type.
- The call "`scanner.position()`" gets the position of the current token. It returns a tuple containing (1) the input file name, (2) the line number, and (3) the column number.

And, here are some comments on constructing the patterns used in a lexicon:

- `Range` constructs a pattern that matches any character in the range.
- `Rep` constructs a pattern that matches a sequence of zero or more items.
- `Rep1` constructs a pattern that matches a sequence of *one* or more items.
- "`pat1 + pat2`" constructs a pattern that matches a sequence containing *pat1* followed by *pat2*.

- "`pat1 | pat2`" constructs a pattern that matches either *pat1* or *pat2*.
- `Any` constructs a pattern that matches any one character in its argument.

Now let's revisit our recursive descent parser, this time with a tokenizer built with **Plex**. The tokenizer is trivial, but will serve as an example of how to hook it into a parser.

```
#!/usr/bin/env python

"""
python_201_rparser_plex.py

A recursive descent parser example.
This example uses Plex to implement a tokenizer.

The grammar:

    Prog ::= Command | Command Prog
    Command ::= Func_call
    Func_call ::= Term '(' Func_call_list ')'
    Func_call_list ::= Func_call | Func_call ',' Func_call_list
    Term = <word>

"""

import sys, string, types
import getopt
import Plex

## from IPython.Shell import IPShellEmbed
## ipshell = IPShellEmbed((),
##     banner = '>>>>>>>> Into IPython >>>>>>>>',
##     exit_msg = '<<<<<<<< Out of IPython <<<<<<<<')

#
# Constants
#

# AST node types
NoneNodeType =         0
ProgNodeType =         1
CommandNodeType =      2
FuncCallNodeType =     3
FuncCallListNodeType = 4
TermNodeType =         5

# Token types
NoneTokType =  0
LParTokType =  1
RParTokType =  2
```

```
WordTokType =  3
CommaTokType = 4
EOFTokType =   5

# Dictionary to map node type values to node type names
NodeTypeDict = {
    NoneNodeType:         'NoneNodeType',
    ProgNodeType:         'ProgNodeType',
    CommandNodeType:      'CommandNodeType',
    FuncCallNodeType:     'FuncCallNodeType',
    FuncCallListNodeType: 'FuncCallListNodeType',
    TermNodeType:         'TermNodeType',
    }

#
# Representation of a node in the AST (abstract syntax tree).
#
class ASTNode:
    def __init__(self, nodeType, *args):
        self.nodeType = nodeType
        self.children = []
        for item in args:
            self.children.append(item)
    def show(self, level):
        self.showLevel(level)
        print 'Node -- Type %s' % NodeTypeDict[self.nodeType]
        level += 1
        for child in self.children:
            if isinstance(child, ASTNode):
                child.show(level)
            elif type(child) == types.ListType:
                for item in child:
                    item.show(level)
            else:
                self.showLevel(level)
                print 'Child:', child
    def showLevel(self, level):
        for idx in range(level):
            print '   ',

#
# The recursive descent parser class.
#   Contains the "recognizer" methods, which implement the grammar
#   rules (above), one recognizer method for each production rule.
#
class ProgParser:
    def __init__(self):
        pass

    def parseFile(self, infileName):
        self.tokens = None
```

```
        self.tokenType = NoneTokType
        self.token = ''
        self.lineNo = -1
        self.infile = file(infileName, 'r')
        self.tokens = genTokens(self.infile, infileName)
        try:
            self.tokenType, self.token, self.lineNo = self.tokens.next()
        except StopIteration:
            raise RuntimeError, 'Empty file'
        result = self.prog_reco()
        self.infile.close()
        self.infile = None
        return result

    def parseStream(self, instream):
        self.tokens = None
        self.tokenType = NoneTokType
        self.token = ''
        self.lineNo = -1
        self.tokens = genTokens(self.instream, '<stream>')
        try:
            self.tokenType, self.token, self.lineNo = self.tokens.next()
        except StopIteration:
            raise RuntimeError, 'Empty stream'
        result = self.prog_reco()
        self.infile.close()
        self.infile = None
        return result

    def prog_reco(self):
        commandList = []
        while 1:
            result = self.command_reco()
            if not result:
                break
            commandList.append(result)
        return ASTNode(ProgNodeType, commandList)

    def command_reco(self):
        if self.tokenType == EOFTokType:
            return None
        result = self.func_call_reco()
        return ASTNode(CommandNodeType, result)

    def func_call_reco(self):
        if self.tokenType == WordTokType:
            term = ASTNode(TermNodeType, self.token)
            self.tokenType, self.token, self.lineNo = self.tokens.next()
            if self.tokenType == LParTokType:
                self.tokenType, self.token, self.lineNo = self.tokens.next(
                result = self.func_call_list_reco()
```

```
                    if result:
                        if self.tokenType == RParTokType:
                            self.tokenType, self.token, self.lineNo = \
                                self.tokens.next()
                            return ASTNode(FuncCallNodeType, term, result)
                        else:
                            raise ParseError(self.lineNo, 'missing right paren'
                    else:
                        raise ParseError(self.lineNo, 'bad func call list')
                else:
                    raise ParseError(self.lineNo, 'missing left paren')
            else:
                return None

        def func_call_list_reco(self):
            terms = []
            while 1:
                result = self.func_call_reco()
                if not result:
                    break
                terms.append(result)
                if self.tokenType != CommaTokType:
                    break
                self.tokenType, self.token, self.lineNo = self.tokens.next()
            return ASTNode(FuncCallListNodeType, terms)

    #
    # The parse error exception class.
    #
    class ParseError(Exception):
        def __init__(self, lineNo, msg):
            RuntimeError.__init__(self, msg)
            self.lineNo = lineNo
            self.msg = msg
        def getLineNo(self):
            return self.lineNo
        def getMsg(self):
            return self.msg

    #
    # Generate the tokens.
    # Usage - example
    #    gen = genTokens(infile)
    #    tokType, tok, lineNo = gen.next()
    #    ...
    def genTokens(infile, infileName):
        letter = Plex.Range("AZaz")
        digit =  Plex.Range("09")
        name = letter +  Plex.Rep(letter | digit)
        lpar = Plex.Str('(')
        rpar = Plex.Str(')')
```

```
        comma = Plex.Str(',')
        comment = Plex.Str("#") + Plex.Rep(Plex.AnyBut("\n"))
        space = Plex.Any(" \t\n")
        lexicon = Plex.Lexicon([
            (name,      'word'),
            (lpar,      'lpar'),
            (rpar,      'rpar'),
            (comma,     'comma'),
            (comment,   Plex.IGNORE),
            (space,     Plex.IGNORE),
        ])
        scanner = Plex.Scanner(lexicon, infile, infileName)
        while 1:
            tokenType, token = scanner.read()
            name, lineNo, columnNo = scanner.position()
            if tokenType == None:
                tokType = EOFTokType
                token = None
            elif tokenType == 'word':
                tokType = WordTokType
            elif tokenType == 'lpar':
                tokType = LParTokType
            elif tokenType == 'rpar':
                tokType = RParTokType
            elif tokenType == 'comma':
                tokType = CommaTokType
            else:
                tokType = NoneTokType
            tok = token
            yield (tokType, tok, lineNo)

    def test(infileName):
        parser = ProgParser()
        #ipshell('(test) #1\nCtrl-D to exit')
        result = None
        try:
            result = parser.parseFile(infileName)
        except ParseError, exp:
            sys.stderr.write('ParseError: (%d) %s\n' % \
                (exp.getLineNo(), exp.getMsg()))
        if result:
            result.show(0)

    USAGE_TEXT = """
    Usage:
        python python_201_rparser_plex.py [options] <inputfile>
    Options:
        -h, --help      Display this help message.
    Example:
        python python_201_rparser_plex.py myfile.txt
    """
```

```
def usage():
    print USAGE_TEXT
    sys.exit(-1)

def main():
    args = sys.argv[1:]
    try:
        opts, args = getopt.getopt(args, 'h', ['help'])
    except:
        usage()
    for opt, val in opts:
        if opt in ('-h', '--help'):
            usage()
    if len(args) != 1:
        usage()
    infileName = args[0]
    test(infileName)

if __name__ == '__main__':
    main()
    #import pdb
    #pdb.run('main()')
```

Download as text (original file name: Examples/python_201_rparser_plex.py).

And, here is a sample of the data we can apply this parser to:

```
# Test for recursive descent parser and Plex.
# Command #1
aaa()
# Command #2
bbb (ccc())    # An end of line comment.
# Command #3
ddd(eee(), fff(ggg(), hhh(), iii()))
# End of test
```

Download as text (original file name: Examples/python_201_rparser_plex_data.txt).

Comments:

- We can now put comments in our input, and they will be ignored. Comments begin with a "#" and continue to the end of line. See the definition of *comment* in function `genTokens`.
- This tokenizer does not require us to separate tokens with whitespace as did the simple tokenizer in the earlier version of our recursive descent parser.
- The changes we made over the earlier version were to:
    - Import `Plex`.

    - Replace the definition of the tokenizer function `genTokens`.
    - Change the call to `genTokens` so that the call passes in the file name, which is needed to create the scanner.
- Our new version of `genTokens` does the following:
    1. Create patterns for scanning.
    2. Create a lexicon (an instance of `Plex.Lexicon`), which uses the patterns.
    3. Create a scanner (an instance of `Plex.Scanner`), which uses the lexicon.
    4. Execute a loop that reads tokens (from the scanner) and "yields" each one.

5.4 A survey of existing tools

For complex parsing tasks, you may want to consider the following tools:

- kwParsing -- A parser generator in Python -- http://www.pythonpros.com/arw/kwParsing/kwParsing.html.
- PLY -- Python Lex-Yacc -- http://systems.cs.uchicago.edu/ply/.
- PyLR -- Fast LR parsing in python -- http://starship.python.net/crew/scott/PyLR.html.
- Yapps -- The Yapps Parser Generator System -- http://theory.stanford.edu/~amitp/Yapps/.

And, for lexical analysis, you may also want to look at -- Using Regular Expressions for Lexical Analysis.

5.5 Creating a parser with PLY

In this section we will show how to implement our parser example with **PLY**.

First down-load **PLY**. It is available at http://systems.cs.uchicago.edu/ply/.

Then add the **PLY** directory to your *PYTHONPATH*.

Learn how to construct lexers and parsers with **PLY** by reading doc/ply.html in the distribution of **PLY** and by looking at the examples in the distribution.

For those of you who want a more complex example, see A Python Parser for the RELAX NG Compact Syntax, which is implemented with **PLY**.

Now, here is our example parser. Comments and explanations are below.

```
#!/usr/bin/env python
```

```
"""
python_201_parser_ply.py

A parser example.
This example uses PLY to implement a lexer and parser.

The grammar:

    Prog ::= Command*
    Command ::= Func_call
    Func_call ::= Term '(' Func_call_list ')'
    Func_call_list ::= Func_call*
    Term = <word>

"""

import sys, types
import getopt
import lex
import yacc

#
# Globals
#

startlinepos = 0

#
# Constants
#

# AST node types
NoneNodeType =         0
ProgNodeType =         1
CommandNodeType =      2
CommandListNodeType =  3
FuncCallNodeType =     4
FuncCallListNodeType = 5
TermNodeType =         6

# Dictionary to map node type values to node type names
NodeTypeDict = {
    NoneNodeType:         'NoneNodeType',
    ProgNodeType:         'ProgNodeType',
    CommandNodeType:      'CommandNodeType',
    CommandListNodeType:  'CommandListNodeType',
    FuncCallNodeType:     'FuncCallNodeType',
    FuncCallListNodeType: 'FuncCallListNodeType',
    TermNodeType:         'TermNodeType',
    }
```

```
#
# Representation of a node in the AST (abstract syntax tree).
#
class ASTNode:
    def __init__(self, nodeType, *args):
        self.nodeType = nodeType
        self.children = []
        for item in args:
            self.children.append(item)
    def append(self, item):
        self.children.append(item)
    def show(self, level):
        self.showLevel(level)
        print 'Node -- Type: %s' % NodeTypeDict[self.nodeType]
        level += 1
        for child in self.children:
            if isinstance(child, ASTNode):
                child.show(level)
            elif type(child) == types.ListType:
                for item in child:
                    item.show(level)
            else:
                self.showLevel(level)
                print 'Value:', child
    def showLevel(self, level):
        for idx in range(level):
            print '   ',

#
# Exception classes
#
class LexerError(Exception):
    def __init__(self, msg, lineno, columnno):
        self.msg = msg
        self.lineno = lineno
        self.columnno = columnno
    def show(self):
        sys.stderr.write('Lexer error (%d, %d) %s\n' % \
            (self.lineno, self.columnno, self.msg))

class ParserError(Exception):
    def __init__(self, msg, lineno, columnno):
        self.msg = msg
        self.lineno = lineno
        self.columnno = columnno
    def show(self):
        sys.stderr.write('Parser error (%d, %d) %s\n' % \
            (self.lineno, self.columnno, self.msg))

#
# Lexer specification
```

```
#
tokens = (
    'NAME',
    'LPAR','RPAR',
    'COMMA',
    )

# Tokens

t_LPAR =   r'\('
t_RPAR =   r'\)'
t_COMMA =  r'\,'
t_NAME =   r'[a-zA-Z_][a-zA-Z0-9_]*'

# Ignore whitespace
t_ignore = ' \t'

# Ignore comments ('#' to end of line)
def t_COMMENT(t):
    r'\#[^\n]*'
    pass

def t_newline(t):
    r'\n+'
    global startlinepos
    startlinepos = t.lexer.lexpos - 1
    t.lineno += t.value.count("\n")

def t_error(t):
    global startlinepos
    msg = "Illegal character '%s'" % (t.value[0])
    columnno = t.lexer.lexpos - startlinepos
    raise LexerError(msg, t.lineno, columnno)

#
# Parser specification
#
def p_prog(t):
    'prog : command_list'
    t[0] = ASTNode(ProgNodeType, t[1])

def p_command_list_1(t):
    'command_list : command'
    t[0] = ASTNode(CommandListNodeType, t[1])

def p_command_list_2(t):
    'command_list : command_list command'
    t[1].append(t[2])
    t[0] = t[1]

def p_command(t):
```

```
    'command : func_call'
    t[0] = ASTNode(CommandNodeType, t[1])

def p_func_call_1(t):
    'func_call : term LPAR RPAR'
    t[0] = ASTNode(FuncCallNodeType, t[1])

def p_func_call_2(t):
    'func_call : term LPAR func_call_list RPAR'
    t[0] = ASTNode(FuncCallNodeType, t[1],  t[3])

def p_func_call_list_1(t):
    'func_call_list : func_call'
    t[0] = ASTNode(FuncCallListNodeType, t[1])

def p_func_call_list_2(t):
    'func_call_list : func_call_list COMMA func_call'
    t[1].append(t[3])
    t[0] = t[1]

def p_term(t):
    'term : NAME'
    t[0] = ASTNode(TermNodeType, t[1])

def p_error(t):
    global startlinepos
    msg = "Syntax error at '%s'" % t.value
    columnno = t.lexer.lexpos - startlinepos
    raise ParserError(msg, t.lineno, columnno)

#
# Parse the input and display the AST (abstract syntax tree)
#
def parse(infileName):
    startlinepos = 0
    # Build the lexer
    lex.lex(debug=1)
    # Build the parser
    yacc.yacc()
    # Read the input
    infile = file(infileName, 'r')
    content = infile.read()
    infile.close()
    try:
        # Do the parse
        result = yacc.parse(content)
        # Display the AST
        result.show(0)
    except LexerError, exp:
        exp.show()
    except ParserError, exp:
```

```
        exp.show()

USAGE_TEXT = """
Usage:
    python python_201_parser_ply.py [options] <inputfile>
Options:
    -h, --help      Display this help message.
Example:
    python python_201_parser_ply.py testfile.prog
"""

def usage():
    print USAGE_TEXT
    sys.exit(-1)

def main():
    args = sys.argv[1:]
    try:
        opts, args = getopt.getopt(args, 'h', ['help'])
    except:
        usage()
    relink = 1
    for opt, val in opts:
        if opt in ('-h', '--help'):
            usage()
    if len(args) != 1:
        usage()
    infileName = args[0]
    parse(infileName)

if __name__ == '__main__':
    main()
    #import pdb
    #pdb.run('main()')
```

Download as text (original file name: Examples/python_201_parser_ply.py).

Applying this parser to the following input:

```
# Test for recursive descent parser and Plex.
# Command #1
aaa()
# Command #2
bbb (ccc())    # An end of line comment.
# Command #3
ddd(eee(), fff(ggg(), hhh(), iii()))
# End of test
```

Download as text (original file name: Examples/python_201_parser_ply_data.txt).

produces the following output:

```
Node -- Type: ProgNodeType
    Node -- Type: CommandListNodeType
        Node -- Type: CommandNodeType
            Node -- Type: FuncCallNodeType
                Node -- Type: TermNodeType
                    Value: aaa
        Node -- Type: CommandNodeType
            Node -- Type: FuncCallNodeType
                Node -- Type: TermNodeType
                    Value: bbb
                Node -- Type: FuncCallListNodeType
                    Node -- Type: FuncCallNodeType
                        Node -- Type: TermNodeType
                            Value: ccc
        Node -- Type: CommandNodeType
            Node -- Type: FuncCallNodeType
                Node -- Type: TermNodeType
                    Value: ddd
                Node -- Type: FuncCallListNodeType
                    Node -- Type: FuncCallNodeType
                        Node -- Type: TermNodeType
                            Value: eee
                    Node -- Type: FuncCallNodeType
                        Node -- Type: TermNodeType
                            Value: fff
                        Node -- Type: FuncCallListNodeType
                            Node -- Type: FuncCallNodeType
                                Node -- Type: TermNodeType
                                    Value: ggg
                            Node -- Type: FuncCallNodeType
                                Node -- Type: TermNodeType
                                    Value: hhh
                            Node -- Type: FuncCallNodeType
                                Node -- Type: TermNodeType
                                    Value: iii
```

Comments and explanation:

- Creating the syntax tree -- Basically, each rule (1) recognizes a non-terminal, (2) creates a node (possibly using the values from the right-hand side of the rule), and (3) returns the node by setting the value of `t[0]`. A deviation from this is the processing of sequences, discussed below.
- Sequences -- `p_command_list_1` and `p_command_list_1` show how to handle sequences of items. In this case:

    1. `p_command_list_1` recognizes a *command* and creates an instance of `ASTNode` with type *CommandListNodeType* and adds the command to it as a child, and
    2. `p_command_list_2` recognizes an additional *command* and adds it (as a child) to the instance of `ASTNode` that represents the list.

- Distinguishing between different forms of the same rule -- In order to process alternatives to the same production rule differently, we use different functions with different implementations. For example, we use:
  - `p_func_call_1` to recognize and process "`func_call : term LPAR RPAR`" (a function call *without* arguments), and
  - `p_func_call_2` to recognize and process "`func_call : term LPAR func_call_list RPAR`" (a function call *with* arguments).
- Reporting errors -- Our parser reports the first error and quits. We've done this by raising an exception when we find an error. We implement two exception classes: `LexerError` and `ParserError`. Implementing more than one exception class enables us to distinguish between different classes of errors (note the multiple `except:` clauses on the `try:` statement in function `parse`). And, we use an instance of the exception class as a container in order to "bubble up" information about the error (e.g. a message, a line number, and a column number).

5.6 Creating a parser with pyparsing

pyparsing is a relatively new parsing package for Python. It was implemented and is supported by Paul McGuire and it shows promise. It appears especially easy to use and seems especially appropriate in particular for quick parsing tasks, although it has features that make some complex parsing tasks easy. It follows a very natural Python style for constructing parsers.

Good documentation comes with the **pyparsing** distribution. See file HowToUseParsing.html. So, I won't try to repeat that here. What follows is an attempt to provide several quick examples to help you solve simple parsing tasks as quickly as possible.

You will also want to look at the samples in the examples directory, which are very helpful. My examples below are fairly simple. You can see more of the ability of **pyparsing** to handle complex tasks in the examples.

Where to get it - You can find **pyparsing** at: http://pyparsing.sourceforge.net/.

How to install it - Put the pyparsing module somewhere on your *PYTHONPATH*.

And now, here are a few examples.

5.6.1 Parsing comma-delimeted lines

Here is a simple grammar for lines containing fields separated by commas:

```
import sys
from pyparsing import alphanums, ZeroOrMore, Word
```

```
fieldDef = Word(alphanums)
lineDef = fieldDef + ZeroOrMore("," + fieldDef)

args = sys.argv[1:]
if len(args) != 1:
    print 'usage: python pyparsing_test1.py <datafile.txt>'
    sys.exit(-1)
infilename = sys.argv[1]
infile = file(infilename, 'r')
for line in infile:
    fields = lineDef.parseString(line)
    print fields
```

Notes and explanation:

- Note how the grammar is constructed from normal Python calls to function and object/class constructors. I've constructed the parser in-line because my examples are simple, but constructing the parser in a function or even a module might make sense for more complex grammars. **pyparsing** makes it easy to use these these different styles.
- Use "+" to specify a sequence. In our example, a *lineDef* is a *fieldDef* followed by ....
- Use `ZeroOrMore` to specify repetition. In our example, a *lineDef* is a *fieldDef* followed by zero or more occurances of comma and *fieldDef*. There is also `OneOrMore` when you want to require at least one occurance.
- Parsing comma delimited text happens so frequently that **pyparsing** provides a shortcut. Replace:

  ```
  lineDef = fieldDef + ZeroOrMore("," + fieldDef)
  ```

  with:

  ```
  lineDef = delimitedList(fieldDef)
  ```

  And note that `delimitedList` takes an optional argument *delim* used to specify the delimiter. The default is a comma.

5.6.2 Parsing functors

This example parses expressions of the form ``func(arg1, arg2, arg3)''.

```
from pyparsing import Word, alphas, alphanums, nums, ZeroOrMore, Literal

lparen = Literal("(")
rparen = Literal(")")
identifier = Word(alphas, alphanums + "_")
integer  = Word( nums )
```

```
functor = identifier
arg = identifier | integer
args = arg + ZeroOrMore("," + arg)
expression = functor + lparen + args + rparen

content = raw_input("Enter an expression: ")
parsedContent = expression.parseString(content)
print parsedContent
```

Explanation:

- Use `Literal` to specify a fixed string that is to be matched exactly. In our example, a *lparen* is a ``(``.
- `Word` takes an optional second argument. With a single (string) argument, it matches any contiguous word made up of characters in the string. With two (string) arguments it matches a word whose first character is in the first string and whose remaining characters are in the second string. So, our definition of *identifier* matches a word whose first character is an alpha and whose remaining characters are alpha-numerics or underscore. As another example, you can think of `Word("0123456789")` as analogous to a regexp containing the pattern `"[0-9]+"`.
- Use a vertical bar for alternation. In our example, an *arg* can be either an *identifier* or an *integer*.

5.6.3 Parsing names, phone numbers, etc.

This example parses expressions having the following form:

```
Input format:

[name]         [phone]       [city, state zip]

Last, first    111-222-3333  city, ca 99999
```

Here is the parser:

```
import sys
from pyparsing import alphas, nums, ZeroOrMore, Word, Group, Suppress, Comb

lastname = Word(alphas)
firstname = Word(alphas)
city = Group(Word(alphas) + ZeroOrMore(Word(alphas)))
state = Word(alphas, exact=2)
zip = Word(nums, exact=5)

name = Group(lastname + Suppress(",") + firstname)
phone = Combine(Word(nums, exact=3) + "-" + Word(nums, exact=3) + "-" + Wor
location = Group(city + Suppress(",") + state + zip)
```

```
record = name + phone + location

args = sys.argv[1:]
if len(args) != 1:
    print 'usage: python pyparsing_test3.py <datafile.txt>'
    sys.exit(-1)
infilename = sys.argv[1]
infile = file(infilename, 'r')
for line in infile:
    line = line.strip()
    if line and line[0] != "#":
        fields = record.parseString(line)
        print fields
```

And, here is some sample input:

```
Jabberer, Jerry          111-222-3333   Bakersfield, CA 95111
Kackler, Kerry           111-222-3334   Fresno, CA 95112
Louderdale, Larry        111-222-3335   Los Angeles, CA 94001
```

Here is output from parsing the above input:

```
[['Jabberer', 'Jerry'], '111-222-3333', [['Bakersfield'], 'CA', '95111']]
[['Kackler', 'Kerry'], '111-222-3334', [['Fresno'], 'CA', '95112']]
[['Louderdale', 'Larry'], '111-222-3335', [['Los', 'Angeles'], 'CA', '94001
```

Comments:

- We use the ``len=n'' argument to the `Word` constructor to restict the parser to accepting a specific number of characters, for example in the zip code and phone number. `Word` also accepts ``min=n'' and ``max=n'' to enable you to restrict the length of a word to within a range.
- We use `Group` to group the parsed results into sub-lists, for example in the definition of city and name. `Group` enables us to organize the parse results into simple parse trees.
- We use `Combine` to join parsed results back into a single string. For example, in the phone number, we can require dashes and yet join the results back into a single string.
- We use `Suppress` to remove unneeded sub-elements from parsed results. For example, we do not need the comma between last and first name.

5.6.4 A more complex example

This example (thanks to Paul McGuire) parses a more complex structure and produces a dictionary.

Here is the code:

```
from pyparsing import Literal, Word, Group, Dict, ZeroOrMore, alphas, nums,
    delimitedList

import pprint

testData = """
+-------+------+------+------+------+------+------+------+------+
|       |  A1  |  B1  |  C1  |  D1  |  A2  |  B2  |  C2  |  D2  |
+=======+======+======+======+======+======+======+======+======+
| min   |   7  |  43  |   7  |  15  |  82  |  98  |   1  |  37  |
| max   |  11  |  52  |  10  |  17  |  85  | 112  |   4  |  39  |
| ave   |   9  |  47  |   8  |  16  |  84  | 106  |   3  |  38  |
| sdev  |   1  |   3  |   1  |   1  |   1  |   3  |   1  |   1  |
+-------+------+------+------+------+------+------+------+------+
"""

# Define grammar for datatable
heading = (Literal(
"+-------+------+------+------+------+------+------+------+------+") +
"|       |  A1  |  B1  |  C1  |  D1  |  A2  |  B2  |  C2  |  D2  |" +
"+=======+======+======+======+======+======+======+======+======+").suppre

vert = Literal("|").suppress()
number = Word(nums)
rowData = Group( vert + Word(alphas) + vert + delimitedList(number,"|") +
vert )
trailing = Literal(
"+-------+------+------+------+------+------+------+------+------+").suppre

datatable = heading + Dict( ZeroOrMore(rowData) ) + trailing

# Now parse data and print results
data = datatable.parseString(testData)
print "data:", data
print "data.asList():",
pprint.pprint(data.asList())
print "data keys:", data.keys()
print "data['min']:", data['min']
print "data.max:", data.max
```

Notes:

- Note the use of `Dict` to create a dictionary. The `print` statements show how to get at the items in the dictionary.
- Note how we can also get the parse results as a list by using method `asList`.
- Again, we use `suppress` to remove unneeded items from the parse results.

6. GUI Applications

6.1 Introduction

This section will help you to put a GUI (graphical user interface) in your Python program.

We will use a particular GUI library: **PyGTK**. We've chosen this because it is reasonably light-weight and our goal is to embed light-weight GUI interfaces in an (possibly) existing application.

For simpler GUI needs, consider **EasyGUI**, which is also described below.

For more heavy-weight GUI needs (for example, complete GUI applications), you may want to explore **WxPython**. See the WxPython home page at: http://www.wxpython.org/

6.2 PyGtk

Information about **PyGTK** is at:

- The **PyGTK** home page
- **PyGTK** at freshmeat.net

6.2.1 A simple message dialog box

In this section we explain how to pop up a simple dialog box from your Python application.

To do this, do the following:

1. Import `gtk` into your Python module.
2. Define the dialog and its behavior.
3. Create an instance of the dialog.
4. Run the event loop.

Here is a sample that displays a message box:

```
#!/usr/bin/env python

import sys
import getopt
import gtk

class MessageBox(gtk.Dialog):
    def __init__(self, message="", buttons=(), pixmap=None,
            modal= gtk.TRUE):
```

```
            gtk.Dialog.__init__(self)
            self.connect("destroy", self.quit)
            self.connect("delete_event", self.quit)
            if modal:
                self.set_modal(gtk.TRUE)
            hbox = gtk.HBox(spacing=5)
            hbox.set_border_width(5)
            self.vbox.pack_start(hbox)
            hbox.show()
            if pixmap:
                self.realize()
                pixmap = Pixmap(self, pixmap)
                hbox.pack_start(pixmap, expand=gtk.FALSE)
                pixmap.show()
            label = gtk.Label(message)
            hbox.pack_start(label)
            label.show()
            for text in buttons:
                b = gtk.Button(text)
                b.set_flags(gtk.CAN_DEFAULT)
                b.set_data("user_data", text)
                b.connect("clicked", self.click)
                self.action_area.pack_start(b)
                b.show()
            self.ret = None
        def quit(self, *args):
            self.hide()
            self.destroy()
            gtk.mainquit()
        def click(self, button):
            self.ret = button.get_data("user_data")
            self.quit()

    # create a message box, and return which button was pressed
    def message_box(title="Message Box", message="", buttons=(), pixmap=None,
            modal= gtk.TRUE):
        win = MessageBox(message, buttons, pixmap=pixmap, modal=modal)
        win.set_title(title)
        win.show()
        gtk.mainloop()
        return win.ret

    def test():
        result = message_box(title='Test #1',
            message='Here is your message',
            buttons=('Ok', 'Cancel'))
        print 'result:', result

    USAGE_TEXT = """
    Usage:
        python simple_dialog.py [options]
```

```
Options:
    -h, --help      Display this help message.
Example:
    python simple_dialog.py
"""

def usage():
    print USAGE_TEXT
    sys.exit(-1)

def main():
    args = sys.argv[1:]
    try:
        opts, args = getopt.getopt(args, 'h', ['help'])
    except:
        usage()
    relink = 1
    for opt, val in opts:
        if opt in ('-h', '--help'):
            usage()
    if len(args) != 0:
        usage()
    test()

if __name__ == '__main__':
    main()
    #import pdb
    #pdb.run('main()')
```

Download as text (original file name: Examples/python_201_gui_message_box.py).

Some explanation:

- First, we import `gtk`
- Next we define a class `MessageBox` that implements a message box. Here are a few important things to know about that class:
  - It is a subclass of `gtk.Dialog`.
  - It creates a label and packs it into the dialog's client area. Note that a `Dialog` is a `Window` that contains a *vbox* at the top of and an *action_area* at the bottom of its client area. The intension is for us to pack miscellaneous widgets into the *vbox* and to put buttons such as "Ok", "Cancel", etc into the *action_area*.
  - It creates one button for each button label passed to its constructor. The buttons are all connected to the `click` method.
  - The `click` method saves the value of the *user_data* for the button that was clicked. In our example, this value will be either "Ok" or "Cancel".

- And, we define a function (`message_box`) that (1) creates an instance of the `MessageBox` class, (2) sets its title, (3) shows it, (4) starts its event loop so that it can get and process events from the user, and (5) returns the result to the caller (in this case "Ok" or "Cancel").
- Our testing function (`test`) calls function `message_box` and prints the result.
- This looks like quite a bit of code, until you notice that the class `MessageBox` and the function `message_box` could be put it a utility module and reused.

6.2.2 A simple text input dialog box

And, here is an example that displays an text input dialog:

```
#!/usr/bin/env python

import sys
import getopt
import gtk

class EntryDialog( gtk.Dialog):
    def __init__(self, message="", default_text='', modal= gtk.TRUE):
        gtk.Dialog.__init__(self)
        self.connect("destroy", self.quit)
        self.connect("delete_event", self.quit)
        if modal:
            self.set_modal(gtk.TRUE)
        box = gtk.VBox(spacing=10)
        box.set_border_width(10)
        self.vbox.pack_start(box)
        box.show()
        if message:
            label = gtk.Label(message)
            box.pack_start(label)
            label.show()
        self.entry = gtk.Entry()
        self.entry.set_text(default_text)
        box.pack_start(self.entry)
        self.entry.show()
        self.entry.grab_focus()
        button = gtk.Button("OK")
        button.connect("clicked", self.click)
        button.set_flags(gtk.CAN_DEFAULT)
        self.action_area.pack_start(button)
        button.show()
        button.grab_default()
        button = gtk.Button("Cancel")
        button.connect("clicked", self.quit)
        button.set_flags(gtk.CAN_DEFAULT)
        self.action_area.pack_start(button)
```

```
            button.show()
            self.ret = None
        def quit(self, w=None, event=None):
            self.hide()
            self.destroy()
            gtk.mainquit()
        def click(self, button):
            self.ret = self.entry.get_text()
            self.quit()

    def input_box(title="Input Box", message="", default_text='',
            modal= gtk.TRUE):
        win = EntryDialog(message, default_text, modal=modal)
        win.set_title(title)
        win.show()
        gtk.mainloop()
        return win.ret

    def test():
        result = input_box(title='Test #2',
            message='Enter a valuexxx:',
            default_text='a default value')
        if result is None:
            print 'Canceled'
        else:
            print 'result: "%s"' % result

    USAGE_TEXT = """
    Usage:
        python simple_dialog.py [options]
    Options:
        -h, --help      Display this help message.
    Example:
        python simple_dialog.py
    """

    def usage():
        print USAGE_TEXT
        sys.exit(-1)

    def main():
        args = sys.argv[1:]
        try:
            opts, args = getopt.getopt(args, 'h', ['help'])
        except:
            usage()
        relink = 1
        for opt, val in opts:
            if opt in ('-h', '--help'):
                usage()
        if len(args) != 0:
```

```
            usage()
        test()

if __name__ == '__main__':
    main()
    #import pdb
    #pdb.run('main()')
```

Download as text (original file name: Examples/python_201_gui_input_box.py).

Most of the explanation for the message box example is relevant to this example, too. Here are some differences:

- Our `EntryDialog` class constructor creates instance of `gtk.Entry`, sets its default value, and packs it into the client area.
- The constructor also automatically creates two buttons: "OK" and "Cancel". The "OK" button is connect to the `click` method, which saves the value of the entry field. The "Cancel" button is connect to the `quit` method, which does *not* save the value.
- And, if class `EntryDialog` and function `input_box` look usable and useful, add them to your utility gui module.

6.2.3 A file selection dialog box

This example shows a file selection dialog box:

```
#!/usr/bin/env python

import sys
import getopt
import gtk

class FileChooser(gtk.FileSelection):
    def __init__(self, modal=gtk.TRUE, multiple=gtk.TRUE):
        gtk.FileSelection.__init__(self)
        self.multiple = multiple
        self.connect("destroy", self.quit)
        self.connect("delete_event", self.quit)
        if modal:
            self.set_modal(gtk.TRUE)
        self.cancel_button.connect('clicked', self.quit)
        self.ok_button.connect('clicked', self.ok_cb)
        if multiple:
            self.set_select_multiple(gtk.TRUE)
##         self.hide_fileop_buttons()
        self.ret = None
    def quit(self, *args):
        self.hide()
```

```
            self.destroy()
            gtk.mainquit()
        def ok_cb(self, b):
            if self.multiple:
                self.ret = self.get_selections()
            else:
                self.ret = self.get_filename()
            self.quit()

    def file_sel_box(title="Browse", modal=gtk.FALSE, multiple=gtk.TRUE):
        win = FileChooser(modal=modal, multiple=multiple)
        win.set_title(title)
        win.show()
        gtk.mainloop()
        return win.ret

    def file_open_box(modal=gtk.TRUE):
        return file_sel_box("Open", modal=modal, multiple=gtk.TRUE)
    def file_save_box(modal=gtk.TRUE):
        return file_sel_box("Save As", modal=modal, multiple=gtk.FALSE)

    def test():
        result = file_open_box()
        print 'open result:', result
        result = file_save_box()
        print 'save result:', result

    USAGE_TEXT = """
    Usage:
        python simple_dialog.py [options]
    Options:
        -h, --help      Display this help message.
    Example:
        python simple_dialog.py
    """

    def usage():
        print USAGE_TEXT
        sys.exit(-1)

    def main():
        args = sys.argv[1:]
        try:
            opts, args = getopt.getopt(args, 'h', ['help'])
        except:
            usage()
        relink = 1
        for opt, val in opts:
            if opt in ('-h', '--help'):
                usage()
        if len(args) != 0:
```

```
            usage()
        test()

    if __name__ == '__main__':
        main()
        #import pdb
        #pdb.run('main()')
```

Download as text (original file name: Examples/python_201_gui_fileselect.py).

A little guidance:

- There is a pre-defined file selection dialog. We sub-class it.
- This example displays the file selection dialog twice: once with a title "Open" and once with a title "Save As".
- Note how we can control whether the dialog allows multiple file selections. And, if we select the multiple selection mode, then we use `get_selections` instead of `get_filename` in order to get the selected file names.
- The dialog contains buttons that enable the user to (1) create a new folder, (2) delete a file, and (3) rename a file. If you do *not* want the user to perform these operations, then call `hide_fileop_buttons`. This call is commented out in our sample code.

Note that there are also predefined dialogs for font selection (`FontSelectionDialog`) and color selection (`ColorSelectionDialog`)

6.3 EasyGUI

If your GUI needs are minimalist and your application is imperative rather than event driven, then you may want to consider **EasyGUI**. As the name suggests, it is extremely easy to use.

How to know when you might be able to use **EasyGUI**:

- You do *not* need menus.
- Your GUI needs amount to little more than displaying a dialog now and then to get responses from the user.
- You do *not* want to write an event driven application, than is one in which your code sits and waits for the the user to initiate operation, for example, with menu items.

EasyGUI is available at http://www.ferg.org/easygui/.

Information about **EasyGUI** is provided in a text file in the distribution.

EasyGUI provides functions for a variety of commonly needed dialog boxes, including:

- A message box displays a message.
- A yes/no message box displays "Yes" and "No" buttons.
- A continue/cancel message box displays "Continue" and "Cancel" buttons.
- A choice box displays a selection list.
- An enter box allows entry of a line of text.
- An integer box allows entry of an interger.
- A multiple entry box allows entry into multiple fields.
- Code and text boxes support the display of text in monospaced or porportional fonts.
- File and directory boxes enable the user to select a file or a directory.

6.3.1 A simple EasyGUI example

Here is a simple example that prompts the user for an entry, then shows the response in a message box:

```
def testeasygui():
    response = easygui.enterbox(message='Enter your name:',
        title='Name Entry')
    easygui.msgbox(message=response,
        title='Your Response',
        )
```

7. Guidance on Packages and Modules

7.1 Introduction

Python has an excellent range of implementation organization structures. These range from statements and control structures (at a low level) through functions, methods, and classes (at an intermediate level) and modules and packages at an upper level.

This section provides some guidance with the use of packages. In particular:

- How to construct and implement them.
- How to use them.
- How to distribute and install them.

7.2 Implementing Packages

A Python package is a collection of Python modules in a disk directory.

In order to be able to import individual modules from a directory, the directory must contain a file named __init__.py. (Note that requirement does not apply to directories that on listed in PYTHONPATH.) The __init__.py serves several purposes:

- The presence of the file __init__.py in a directory marks the directory as a Python package, which enables importing modules from the directory.
- The first time an application imports any module from the directory/package, the code in the module `__init__` is evaluated.
- If the package itself is imported (as opposed to an individual module within the directory/package), then it is the `__init__` that is imported (and evaluated).

7.3 Using Packages

One simple way to enable the user to import and use a package is to instruct the use to import individual modules from the package.

A second, slightly more advanced way to enable the user to import the package is to expose those features of the package in the `__init__` module. Suppose that module `mod1` contains functions `fun1a` and `fun1b` and suppose that module `mod2` contains functions `fun2a` and `fun2b`. Then file __init__.py might contain the following:

```
from mod1 import fun1a, fun1b
from mod2 import fun2a, fun2b
```

Download as text (original file name: Examples/Testpackages/testpackages/__init__.py).

Then, if the following is evaluated in the user's code:

```
import testpackages
```

Then *testpackages* will contain `fun1a`, `fun1b`, `fun2a`, and `fun2b`.

For example, here is an interactive session that demostrates importing the package:

```
>>> import testpackages
>>> print dir(testpackages)
[`__builtins__', `__doc__', `__file__', `__name__', `__path__',
`fun1a', `fun1b', `fun2a', `fun2b', `mod1', `mod2']
```

7.4 Distributing and Installing Packages

Distutils (Python Distribution Utilities) has special support for distrubuting and

installing packages.

In this section we'll learn how to use **Distutils** to package and install a distribution that contains a single package with multiple modules.

As our example, imagine that we have a directory containing the following:

- Testpackages
- Testpackages/README
- Testpackages/MANIFEST.in
- Testpackages/setup.py
- Testpackages/testpackages/__init__.py
- Testpackages/testpackages/mod1.py
- Testpackages/testpackages/mod2.py

Notice the sub-directory Testpackages/testpackages containing the file __init__.py. This is the Python package that we will install.

We'll describe how to configure the above files so that they can be packaged as a single distribution file and so that the Python package they contain can be installed as a package by **Distutils**.

The MANIFEST.in file lists the files that we want included in our distribution. Here is the contents of our MANIFEST.in file:

```
include README MANIFEST MANIFEST.in
include setup.py
include testpackages/*.py
```

Download as text (original file name: Examples/Testpackages/MANIFEST.in).

The setup.py describes to **Distutils** (1) how to package the distribution file and (2) how to install the distribution. Here is the contents of our sample setup.py:

```
#!/usr/bin/env python

from distutils.core import setup                    # [1]

long_description = 'Tests for installing and distributing Python packages'

setup(name = 'testpackages',                        # [2]
    version = '1.0a',
    description = 'Tests for Python packages',
    maintainer = 'Dave Kuhlman',
    maintainer_email = 'dkuhlman@rexx.com',
    url = 'http://www.rexx.com/ dkuhlman',
    long_description = long_description,
    packages = ['testpackages']                     # [3]
    )
```

Download as text (original file name: Examples/Testpackages/setup.py).

Explanation:

1. We import the necessary component from **Distutils**.
2. We describe the package and its developer/maintainer.
3. We specify the directory that is to be installed as a package. When the user installs our distribution, this directory and all the modules in it will be installed as a package.

Now, to create a distribution file, we run the following:

```
python setup.py sdist --formats=gztar
```

which will create a file testpackages-1.0a.tar.gz under the directory dist.

Then, the user, who wishes to install this file, can do so by executing the following:

```
tar xvzf testpackages-1.0a.tar.gz
cd testpackages-1.0a
python setup.py build
python setup.py install        # as root
```

End Matter

Acknowledgements and Thanks

Thanks to the implementors of **Python** for producing an exceptionally usable and enjoyable programming language.

Thanks to Dave Beazley and others for **SWIG** and **PLY**.

Thanks to Greg Ewing for **Pyrex** and **Plex**.

Thanks to James Henstridge for **PyGTK**.

See Also:

The main Python Web Site
for more information on Python

Python Documentation
for lots of documentation on Python

The Python XML Special Interest Group
for more information on processing XML with Python

Dave's Web Site
for more software and information on using Python for XML and the Web

The SWIG home page
for more information on SWIG (Simplified Wrapper and Interface Generator)

The Pyrex home page
for more information on Pyrex

PLY (Python Lex-Yacc) home page
for more information on PLY

The Plex home page
for more information on Plex

*The **Distutils** documentation at the Python site*
for more information on **Distutils**

About this document ...

Python 201 -- (Slightly) Advanced Python Topics, June 6, 2003, Release 1.00

This document was generated using the **LaTeX**2HTML translator.

LaTeX2HTML is Copyright © 1993, 1994, 1995, 1996, 1997, Nikos Drakos, Computer Based Learning Unit, University of Leeds, and Copyright © 1997, 1998, Ross Moore, Mathematics Department, Macquarie University, Sydney.

The application of **LaTeX**2HTML to the Python documentation has been heavily tailored by Fred L. Drake, Jr. Original navigation icons were contributed by Christopher Petrilli.

Release 1.00, documentation updated on June 6, 2003.