OpenMath support in Python
Python project 1

130016030, 130009542, 130013767
University of St Andrews
March 10, 2015

Introduction

The aim of this project is to implement a basic support of the OpenMath (a standard for representing mathematical objects) encoding in Python. This implementation could be used to store and exchange mathematical objects between different platforms and applications. This involves implementing bi-directional support for OpenMath objects and content dictionaries, using an XML parser.

Running instructions

This project is using builder called PyBuilder. First of all, you need to install it by typing a commands:

git clone https://github.com/aelgru/fluentmock

cd fluentmock

virtualenv venv; source venv/bin/activate

pip install pybuilder

pyb installdependencies

pyb

Otherwise, you can always just forget about that and use Python interpreter by typing:

python "namefile"

Design and implementation

1 omparse.py

This file contains functions that parse XML Open Math file and those functions return Python expressions. It is recursive descend parser, so the every expression is parsed by ParseOMelement function. This parser has been extended, so it could parse more elements of the OpenMath. All suggested extensions were done. For the arithmetic operations it has been decided to pass operand as an argument. This decision simplifies and avoids code duplication.

2 omput.py

This file contains functions that generates the XML tag for the OpenMath symbol/element. It builds the XML tree recursively. Every element is parsed by function OMelement and recognises by comparing the types. This file uses xml.etree.ElementTree and numpy libraries to construct XML DOM tree and parse the matrix.

3 openmath.py

Contains some higher level functionality. Using methods from the parsing file, ParseOMfile and Parse OMString are used to parse strings and files containing OpenMath objects. OMprint and OMstring are used to print or return a string with the OpenMath encoding of the given object. OMprint was modified using minidom to implement pretty-printing for OpenMath objects.

4 errors.py

This file has a list that contains OpenMath objects represented as strings. The OpenMath objects contains deliberate errors. This is done to test this project for all exception handling instead of using one file demo.py.

5 demo.py

Demo is used for an interactive demonstration of our OpenMath encoding. We initially parse strings containing OpenMath objects and arithmetic operations (represented in XML) using the methods provided in openmath.py.

To demonstrate that our implementation works, we parse the XML files provided in the tst subdirectory. An XML file to represent a Python dictionary was also created to test our implementation of a content dictionary to encode Python dictionaries.

All the different objects we support are then returned as strings and also printed with the OpenMath encoding.

Finally, we produce OpenMath encoding for each of our objects and then parse it and check that the result of the parsing is equal to the original object. This demonstrates the correct implementation of bi-directional support for objects.

6 simplify.py

Simplify takes in two arguments, the input and output file, similar to how sockets is handled. The input file contains the OpenMath document to get simplified, and the output file is where the simplified document gets saved too.

It simplifies documents by first converting them from OpenMath into a representation in Python. As such, Python then combines values naturally by itself. When ran back through the parser to be converted into OpenMath, the representation of the original document has been compressed, as those values which could of been combined, have been combined. We perform various checks, namely if the inputfile exists. In which case we provide an appropriate error message to the user.

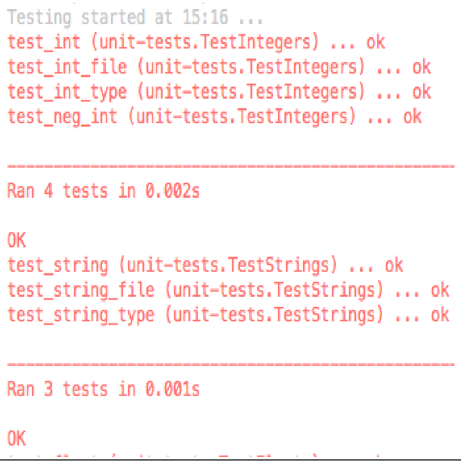
7 client.py and server.py

This extension is composed of two files: server.py and client.py. The server.py file is ran initially to open a socket and listen in for incoming clients. For ease of use by lesser experienced users, the server and client default to the localhost and port 50007. However, these can be changed through passing command line arguments to server.py as documented in README.md. The server can accept a client one after the other, but doesn't queue clients nor implement multiple threading. Once the server receives OpenMath objects from the client it calls ParseOMstring to parse the data, then returns the result back to the client.

The client is also able to (optionally) supply a desired hostname and port number. It is required, however, to at least give the input file (OpenMath doc to parse) and the output file (where to save the result to). We could have left the user to redirect the standard output to a file using redirection but it would mean that we would not be able to supply information on screen that wasn't intended for the standard error stream.

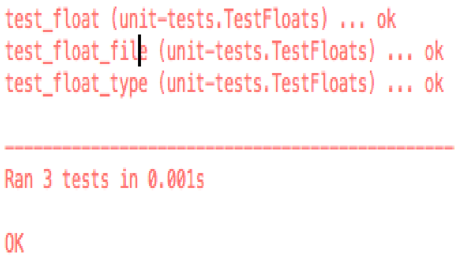
Testing

Unit tests were written for all the different OpenMath objects and symbols that are represented, to confirm that they are being parsed from files and strings and output correctly as the aim is bi-directional support.



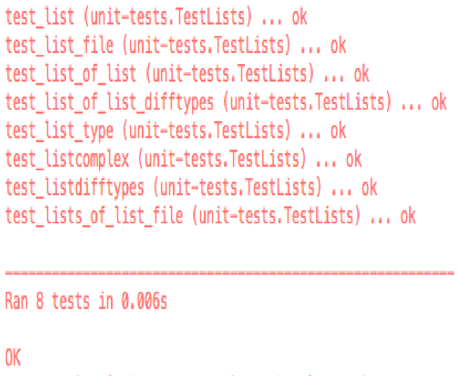
```
Testing started at 15:16 ...
test_int (unit-tests.TestIntegers) ... ok
test_int_file (unit-tests.TestIntegers) ... ok
test_int_type (unit-tests.TestIntegers) ... ok
test_neg_int (unit-tests.TestIntegers) ... ok

----------------------------------------------------------------------
Ran 4 tests in 0.002s

OK
test_string (unit-tests.TestStrings) ... ok
test_string_file (unit-tests.TestStrings) ... ok
test_string_type (unit-tests.TestStrings) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.001s

OK
```

For the basic types, such as integer and string, OpenMath encoding for an object is produced, then parsed and checked that the result of the parsing is equal to the original object. There is a check that parsing an XML file will produce the correct object and that the Python type matches so that an integer isn't being returned in string form for example. For the type tests, strings are used instead of files to make sure that strings are parsing correctly too.



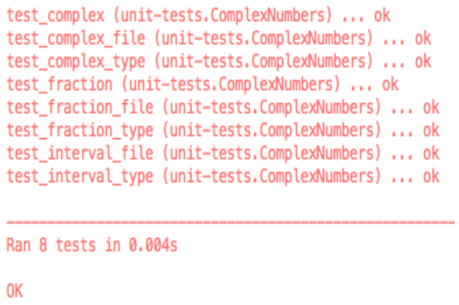
```
test_float (unit-tests.TestFloats) ... ok
test_float_file (unit-tests.TestFloats) ... ok
test_float_type (unit-tests.TestFloats) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.001s

OK
```

For floats, a slightly different approach is taken. As opposed to separating all the possible forms of floats into different tests, they are tested in one unit test, in a list. Other than this change, they are tested in the same way as strings and integers.



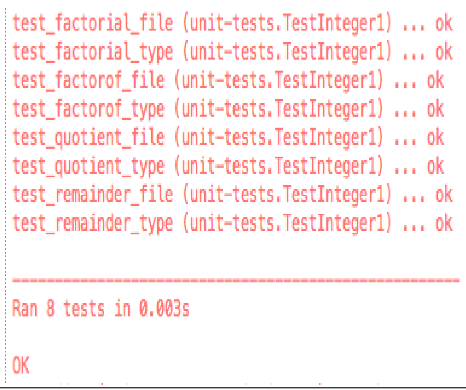
```
test_list (unit-tests.TestLists) ... ok
test_list_file (unit-tests.TestLists) ... ok
test_list_of_list (unit-tests.TestLists) ... ok
test_list_of_list_difftypes (unit-tests.TestLists) ... ok
test_list_type (unit-tests.TestLists) ... ok
test_listcomplex (unit-tests.TestLists) ... ok
test_listdifftypes (unit-tests.TestLists) ... ok
test_lists_of_list_file (unit-tests.TestLists) ... ok

----------------------------------------------------------------------
Ran 8 tests in 0.006s

OK
```

Lists are able to include different types of object. So when it came to testing lists and lists of lists, extra tests are included to check that there are no issues with inserting an integer, a string, a boolean and a float in the same list. This test passes and so the tests are taken even further; inserting fractions, complex numbers and lists inside the same list.



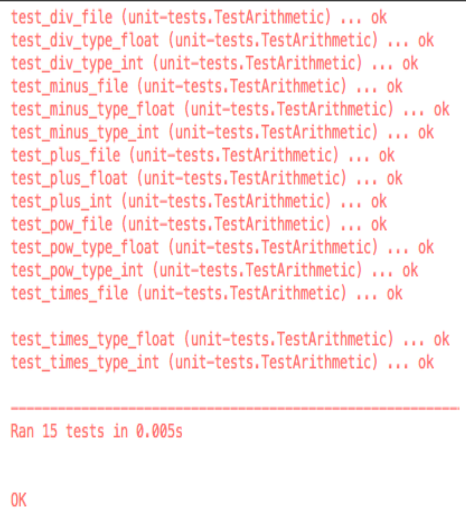
```
test_complex (unit-tests.ComplexNumbers) ... ok
test_complex_file (unit-tests.ComplexNumbers) ... ok
test_complex_type (unit-tests.ComplexNumbers) ... ok
test_fraction (unit-tests.ComplexNumbers) ... ok
test_fraction_file (unit-tests.ComplexNumbers) ... ok
test_fraction_type (unit-tests.ComplexNumbers) ... ok
test_interval_file (unit-tests.ComplexNumbers) ... ok
test_interval_type (unit-tests.ComplexNumbers) ... ok

----------------------------------------------------------------------
Ran 8 tests in 0.004s

OK
```

As rational and complex numbers can only take integers, they are tested in the same basic way as integers and strings because error messages are tested separately (errors.py).



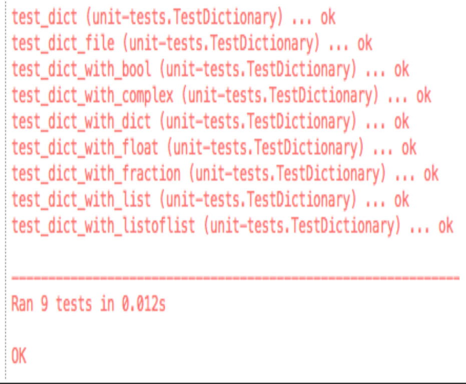
```
test_factorial_file (unit-tests.TestInteger1) ... ok
test_factorial_type (unit-tests.TestInteger1) ... ok
test_factorof_file (unit-tests.TestInteger1) ... ok
test_factorof_type (unit-tests.TestInteger1) ... ok
test_quotient_file (unit-tests.TestInteger1) ... ok
test_quotient_type (unit-tests.TestInteger1) ... ok
test_remainder_file (unit-tests.TestInteger1) ... ok
test_remainder_type (unit-tests.TestInteger1) ... ok

----------------------------------------------------------------------
Ran 8 tests in 0.003s

OK
```

The same goes for everything from the integer1 content dictionary which is tested in the same unit test. As all of these operations (remainder, factor of and quotient) strictly take two integers, they are simply tested for types again and the result expected for each calculation. This is important as all of these calculations should only return integers too.



```
test_div_file (unit-tests.TestArithmetic) ... ok
test_div_type_float (unit-tests.TestArithmetic) ... ok
test_div_type_int (unit-tests.TestArithmetic) ... ok
test_minus_file (unit-tests.TestArithmetic) ... ok
test_minus_type_float (unit-tests.TestArithmetic) ... ok
test_minus_type_int (unit-tests.TestArithmetic) ... ok
test_plus_file (unit-tests.TestArithmetic) ... ok
test_plus_float (unit-tests.TestArithmetic) ... ok
test_plus_int (unit-tests.TestArithmetic) ... ok
test_pow_file (unit-tests.TestArithmetic) ... ok
test_pow_type_float (unit-tests.TestArithmetic) ... ok
test_pow_type_int (unit-tests.TestArithmetic) ... ok
test_times_file (unit-tests.TestArithmetic) ... ok

test_times_type_float (unit-tests.TestArithmetic) ... ok
test_times_type_int (unit-tests.TestArithmetic) ... ok

----------------------------------------------------------------------
Ran 15 tests in 0.005s


OK
```

As all of the arithmetic operations are able to take both integers and floating point numbers, they are tested first with file tests initially mainly to make sure arithmetic operations are producing the correct results. As floats are used in these files, assertEquals can no longer be used as there is no way of precisely representing floats due to their infinite nature; assertAlmostEquals is used instead. String are parsed next with both integers and floats as calculations with two integers should return an integer whereas calculations with even one float should return floats.



```
test_dict (unit-tests.TestDictionary) ... ok
test_dict_file (unit-tests.TestDictionary) ... ok
test_dict_with_bool (unit-tests.TestDictionary) ... ok
test_dict_with_complex (unit-tests.TestDictionary) ... ok
test_dict_with_dict (unit-tests.TestDictionary) ... ok
test_dict_with_float (unit-tests.TestDictionary) ... ok
test_dict_with_fraction (unit-tests.TestDictionary) ... ok
test_dict_with_list (unit-tests.TestDictionary) ... ok
test_dict_with_listoflist (unit-tests.TestDictionary) ... ok

----------------------------------------------------------------------
Ran 9 tests in 0.012s

OK
```

The final unit test is for the implementation of a content dictionary for Python dictionaries. Keys in Python dictionaries may not always be strings and values are any object that may be encoded in OpenMath (including other dictionaries). Therefore, dictionaries are checked with everything; booleans, floats, lists, lists of lists, fractions, complex numbers and other dictionaries.

Individual reports

130016030

My contribution to this project was implementing support for the input for interval and input/output for matrix and dictionary. After that, I did medium requirements, such as: arithmetic operations, factorial. For Medium to Hard requirements I implemented functionality to output OpenMath objects using prescribed content dictionaries and non-trivial operations, namely: root, sum, product, gcd, lcm. Finally, I added support for attributions and errors.

130009542

I first implemented support for strings by analogy as support for integers was already implemented. I then moved on to the first few extensions. I implemented pretty printing for OpenMath objects represented in XML using minidom in the OMprint method as I found that ElementTree does not support pretty printing. I then developed a content dictionary to encode Python dictionaries (by first representing a dictionary in an XML file to use as an example) and worked on arithmetic operations. I also implemented support for some OpenMath symbols of my choice; I chose the rest of the integer1 content dictionary (remainder, factor of and quotient). Finally, I began writing the report as well as unit tests to confirm that all of the objects we implemented parse and output as expected individually and with each other.

130013767

Initially I added support for floating point numbers by following the structure of the implementation of integers. Once I had a grasp of how the parser worked, I was able to provide support for more complex content directories like rationals, and complex cartesian. At this point, I realised that a lot of assumptions were being made about the type of objects that we would be parsing, and thus began adding error checking in the form of assert statements to make sure we were parsing a correctly formed input, which evolved to using try...except blocks (complying to the ask forgiveness not permission method).

As the project progressed we started looking towards more challenging extensions, namely sockets. The extension require a client to be able to send Open Math documents over a network to a server so that they could be parse and evaluated, before being sent back. Afterwards I looked to the simplify script to provide a way to compact OpenMath documents where possible.

Conclusion

To conclude, through implementing a support of the OpenMath encoding in Python, which involved tasks of varying difficulties, we are more familiar and confident with the fundamental features of the Python language. We are comfortable with the basic use of Python, distributing the code of the Python project across multiple files, manipulating Python data types and working with XML parsers. Our encoding bidirectionally supports the basic OpenMath objects (integers, floats and strings), a selection of OpenMath content dictionaries, as well as many of the extension features.

References

1. Kuhlman, D. *A Python Book: Beginning Python, Advanced Python, and Python Exercises*. Platypus Global Media, 2009
2. Downey, A. *Think Python* Sebastopol, CA: O'Reilly Media, Inc, 2012.