Packages in *Macaulay 2*

There are many levels at which one can work in *Macaulay 2*. In increasing order of sophistication we have:

1. typing information in to the M2 session.
2. typing information into a file and using F11 to execute commands in *Macaulay 2*.
3. load a file.
4. load or install a package.

We have now discussed the first three and proceed to packages. These are very similar to just having a file with the functions you need in it with a few key additions.

Packages are good for

- functions someone else in the world might be interested in using,
- debugging,
- testing,
- creating documentation for your functions,
- **publishing your work**,
- and I'm sure several other things I can't think of at the moment.

There is a journal, the *Journal of Software for Algebra and Geometry* www.j-sag.org that publishes short papers combined with packages. This is important for two reasons.

1. This is a fantastic resource for seeing what a well-constructed package looks like. As with any peer-reviewed system, they are not perfect, but they are generally very good and serve as a good model/example as you begin to construct your own package.
2. As you build your own packages, consider publishing. The journal is peer-reviewed and has a well-respected editorial board, so it counts!

The ingredients of a package:

1. Code — your functions, both internal and those exported.
2. Documentation
   (a) *External* documentation must be complete to have an error-free install.
   (b) *Internal* documentation is critical both to yourself and others. It is unlikely that you will be the only person to ever work on "your" package, therefore good internal documentation is absolutely critical. A good example of good internal doc is Binomials.m2. It very likely that at some point 6 months or more passes between times you work on your own package and you've forgotten what you were doing. You will save yourself a lot of time with good internal documentation.
3. Tests — this helps both you and later others make sure your code is working as expected and any changes to *Macaulay 2* have not broken your code.

More on each of these pieces now.

Code To build a package, check out packages in the documentation. However, here is information to get you started. The basic package format is shown with the following "silly" example. Some discussion follows. Consider changing various aspects of the package to things you might want/need. We make suggestions of things to change below.

```
newPackage(
        "FirstPackage",
        Version => "1.0",
        Date => "February 11, 2004",
        Authors => {{Name => "Jane Doe",
                  Email => "doe@math.uiuc.edu",
                  HomePage => "http://www.math.uiuc.edu/~doe/"}},
        Headline => "an example Macaulay2 package",
        DebuggingMode => true
        )

needsPackage"SimpleDoc"

export {firstFunction}

firstFunction = method(TypicalValue => String)
firstFunction ZZ := String => n -> if n == 1 then "Hello World!" else "D'oh!"

beginDocumentation()
doc ///
    Key
        FirstPackage
    Headline
        an example Macaulay2 package
    Description
        Text
            This package is a basic package to be used as an example.
    Caveat
            Still trying to figure this out.
///

doc ///
    Key
        firstFunction
        (firstFunction,ZZ)
    Headline
        a silly first function
    Usage
        f = firstFunction n
    Inputs
        n:ZZ
    Outputs
        f:String
          a silly string, depending on the value of @TT "n"@
    Description
     Text
       Here we show an example.
     Example
       firstFunction 1
       firstFunction 0
```

```
///

TEST ///
    assert ( firstFunction 2 == "D'oh!" )
///

end

You can write anything you want down here... I like to keep examples
as I'm developing here.  Clean it up before submitting for
publication.
```

1. You must save a package as file with nothing but the package in it (you can put anything you want, actually, as long is it comes after an end in the file). **The name of the file must match the name of the package.** So for example, you must save this package as "FirstPackage.m2"
2. The discussion here is just about the overall structure of a package. There are examples of more sophisticated documentation nodes and tests that follow this discussion. Consider using that to help make changes to this document.
3. Debugging code is an important skill. Learning the meaning of error messages and how to use *Macaulay 2*'s debugging features go a long way with this. With this in mind, try introducing errors to the package. Think like a scientist — introduce them one at a time in a way that you know what the error is, and then see what *Macaulay 2* tells you.
   (a) Start with the pre-amble.
   (b) Try not exporting the function — just export nothing.
   (c) Introduce and error into the function. Here we can force with using the command error (check it out in the documentation). This can be useful when developing code. More on this in a moment.
   (d) Now introduce errors into the documentation.
   (e) Finally change the test so that the test is false and see what happens.

   Macaulay 2 has some nice debugging features. We won't go into them all now, but one key concept is that if an error occurs while loading a package, or while running functions from a package, *Macaulay 2* moves into debugging mode which is indicated by a change of the input icon to having two i's — for example ii4:. This mode allows access to locally defined variables in the function where the error occurred. If that function is called by another, access to the higher level function is accessed through the use of the command break. This allows you to experiment and play around with the internal structure of the function. Sometimes you might want to do this even when *Macaulay 2* does not think there is an error. The function error is good for this. In this case you might want *Macaulay 2* to continue the computation after you have had a chance to inspect things and the function "continue" does this.

Documentation Internal documentation is easy, just use two dashes, for example the line before the method, and after and then later are all internal comments as they start with two dashes.

```
-- Cellular decomposition of binomial ideals:
binomialCellularDecomposition = method (Options => {returnCellVars => false, verbose=>true})
binomialCellularDecomposition Ideal := Ideal => o -> I -> (
-- Based on code by Ignacio Ojeda and Mike Stillman
     R := ring I;
     n := numgens R;
     Answer := {};
     L := null;
```

```
   IntersectAnswer := ideal(1_R);
   ToDo := {{{1_R},toList(0..n-1),I}};
   -- Each entry of the ToDoList is a triple:
   -- #0 contains list of variables with respect to which is already saturated
   -- #1 contains variables to be considered for cell variables
   -- #2 is the ideal to decompose
```

Below is basic skeleton and then an example of an external documentation node, from the published package Naughty, using the package SimpleDoc.

The skeleton:

```
doc ///
   Key
   Headline
   Usage
   Inputs
   Outputs
   Consequences
    Item
   Description
    Text
    Code
    Pre
    Example
   Subnodes
   Caveat
   SeeAlso
///
```

An example of a full node for the package itself:

```
doc ///
    Key
        Nauty
    Headline
        Interface to nauty
    Description
        Text
            This package provides an interface from Macaulay2 to many of the functions provided in
            the software nauty by Brendan D. McKay, available at @HREF "http://cs.anu.edu.au/~bdm/na
            The nauty package provides very efficient methods for determining whether
            given graphs are isomorphic, generating all graphs with particular properties,
            generating random graphs, and more.

            Most methods can handle graphs in either the Macaulay2 @TO "Graph"@ type as provided by
            the @TO "EdgeIdeals"@ package or as Graph6 and Sparse6 strings as used by nauty.
            The purpose of this is that graphs stored as strings are greatly more efficient than
            graphs stored as instances of the class @TO "Graph"@.
            (See @TO "Comparison of Graph6 and Sparse6 formats"@.)

            It is recommended to work with graphs represented as strings while using nauty-provided
            methods and then converting the graphs to instances fo the class @TO "Graph"@ for furthe
            (e.g., computing the chromatic number).

            The theoretical underpinnings of nauty are in the paper:
            B. D. McKay, "Practical graph isomorphism," Congr. Numer. 30 (1981), 45--87.
```

```
    SeeAlso
        "Comparison of Graph6 and Sparse6 formats"
        "Example: Checking for isomorphic graphs"
        "Example: Generating and filtering graphs"
///
```

and for a function in the package:

```
doc ///
    Key
        buildGraphFilter
        (buildGraphFilter, HashTable)
        (buildGraphFilter, List)
    Headline
        creates the appropriate filter string for use with filterGraphs and countGraphs
    Usage
        s = buildGraphFilter h
        s = buildGraphFilter l
    Inputs
        h:HashTable
            which describes the properties desired in filtering
        l:List
            which describes the properties desired in filtering
    Outputs
        s:String
            which can be used with @TO "filterGraphs"@ and @TO "countGraphs"@
    Description
        Text
            The @TO "filterGraphs"@ and @TO "countGraphs"@ methods both can use a tremendous number
            which are described by a rather tersely encoded string.  This method builds that string
            in the @TO "HashTable"@ $h$ or the @TO "List"@ $l$.  Any keys which do not exist are sim
            any values which are not valid (e.g., exactly $-3$ vertices) are also ignored.

            The values can either be @TO "Boolean"@ or in @TO "ZZ"@.  @TO "Boolean"@ values are trea
            as expected.  Numerical values are more complicated; we use an example to illustrate how
            can be used, but note that this usage works for all numerically valued keys.

            The key @TT "NumEdges"@ restricts to a specific number of edges in the graph.  If the va
            the integer $n$, then only graphs with @EM "exactly"@ $n$ edges are returned.
        Example
            R = QQ[a..f];
            L = {graph {a*b}, graph {a*b, b*c}, graph {a*b, b*c, c*d}, graph {a*b, b*c, c*d, d*e}};
            s = buildGraphFilter {"NumEdges" => 3};
            filterGraphs(L, s)
        Text
            If the value is the @TO "Sequence"@ $(m,n)$, then all graphs with at least $m$ and at mo
        Example
            s = buildGraphFilter {"NumEdges" => (2,3)};
            filterGraphs(L, s)
        Text
            If the value is the @TO "Sequence"@ $(,n)$, then all graphs with at most $n$ edges are r
        Example
            s = buildGraphFilter {"NumEdges" => (,3)};
            filterGraphs(L, s)
        Text
```

```
            If the value is the @TO "Sequence"@ $(m,)$, then all graphs with at least $m$ edges are
        Example
            s = buildGraphFilter {"NumEdges" => (2,)};
            filterGraphs(L, s)
        Text
            Moreover, the associated key @TT "NegateNumEdges"@, if true, causes the @EM "opposite"@
        Example
            s = buildGraphFilter {"NumEdges" => (2,), "NegateNumEdges" => true};
            filterGraphs(L, s)
        Text
            The following are the boolean options: "Regular", "Bipartite", "Eulerian", "VertexTransi

            The following are the numerical options (recall all have the associate "Negate" option):
            "MinDegree", "MaxDegree", "Radius", "Diameter", "Girth", "NumCycles", "NumTriangles", "G
            "FixedPoints", "Connectivity", "MinCommonNbrsAdj", "MaxCommonNbrsAdj", "MinCommonNbrsNon
    Caveat
            @TT "Connectivity"@ only works for the values $0, 1, 2$ and uses the following definitio
            A graph is $k$-connected if $k$ is the minimum size of a set of vertices whose complemen

            Thus, in order to filter for connected graphs, one must use @TT "{\"Connectivity\" => 0,

            @TT "NumCycles"@ can only be used with graphs on at most $n$ vertices, where $n$ is the
            nauty was compiled, typically $32$ or $64$.
    SeeAlso
        countGraphs
        "Example: Generating and filtering graphs"
        filterGraphs
///
```

This is a particularly good example as it is for a method with multiple implementations and uses most of the structures.

Tests Any good package has a series of asserts to make sure everything is working as expected. For example we include a test from Binomials.m2 and IntegralClosure.m2

From Binomials.m2:

```
TEST ///
R = QQ[c,d,x,y,z,w];
I = ideal(x^3*d^2*w-c*z^2,x^5*y^2-w^7,w^3-z^8,z^2-d*w*x^7)
time bpd = binomialPrimaryDecomposition (I,verbose=>false);
assert (intersect bpd == I)
///
```

From IntegralClosure.m2

```
-- integrally closed test
TEST ///
R = QQ[u,v]/ideal(u+2)
time J = integralClosure (R,Variable => symbol a)
use ring ideal J
assert(ideal J == ideal(u+2))
icFractions R  -- NOT GOOD?
///
```

Now that you have examples of all the pieces, the best thing to do is start building your own package. Use the many examples available, the wiki and the google group. This information is available in the Introductory Lab document.