DataEnvironment

API Documentation

March 31, 2009

Contents

1 Package Classes

1.1 Modules

- **DotData**: Classes and functions pertaining to the DotData class, a structured tabular data object. *(Section 2, p. 3)*

2 Module Classes.DotData

Classes and functions pertaining to the DotData class, a structured tabular data object.

2.1 Functions

AddOrReplaceColumns(*X*, *Cols*)

technical dependency of .aggregate_in method

GraySpec(*k*)

For integer argument k, returns list of k gray-scale colors, increasingly light, linearly in the HSV color space, as web hex triplets, . Technical dependency of .aggregate_in method.

DotDataAggregate(*X*, *On*, *AggFuncDict*=None, *returnsort*=False)

```
used to aggregate a DotData on a set of specified factors, using
specified aggregation functions.

ARGUMENTS:
-- X, DotData record array
-- On, list of column names in X
-- AggFuncDict -- dictionary where:
        -- keys are some (or all) column names of X that are NOT in 'On'
        -- values are functions that can be applied to lists or numpy arrays
-- returnsort, boolean:  because the aggregation function sorts
the original data, there's an argsort that can be returned.  If
this boolean is true the return value is        [A,s] where A is the
aggregated dotdata and s is the sorting permutation on the
original data set.  If false (the default) the return value is just A

Intuitively, this function will aggregate the dataset X on the
columns listed in 'On', so that the resulting aggregate data set
has one record for each unique tuples of values in those columns.
The more factors listed in On argument, the "finer" is the
aggregation, the fewer factors, the "coarser" the aggregation.
For example, if On = ['A','B'], the resulting data set will have
one record for each unique value of pairs (a,b) in X[['A','B']].

The AggFuncDict argument specifies how to aggregate the factors
_not_ listed in 'On' -- the so-called "Off" columns. For example, if
On = ['A','B'] and 'C' is some other column, then AggFuncDict['C']
is the function that will be used to reduce to a single value
the (potentially multiple) values in the 'C' column
corresponding to unique values in the 'A', 'B' columns.
For instance, if
        AggFuncDict['C'] = numpy.mean
then the result will be that the values in the 'C' column
corresponding to a single 'A','B' value will be averaged.

If an "Off" column is _not_ provided as a key in AggFuncDict,
a default aggregator function will be used:  the sum
function for numerical columns, concatenation for string columns.
```

DotDataFromPathList(*PathList*)

Opens dotdatas from a list of dot-data paths, assuming they have disjoint columns and identical numbers of rows; then stacks them horizontally, e.g. adding columns side-by-side, aligning the rows.

Pivot(*X*, *a*, *b*, *Keep*=None)

```
Implements pivoting on dotdatas.
See http://en.wikipedia.org/wiki/Pivot_table for information about pivot tables.

ARGUMENTS:
X -- dotdata
a,b -- columns in X
Keep -- list of columns in X

RETURNS:
--X pivoted on (a,b) with a as the row axis and b values
as the column axis.

So-called "nontrivial columns relative to b" in X are added as
color-grouped sets of columns, and "trivial columns relative to b"
are also retained as cross-grouped sets of columns if they're
listed in 'Keep' argument.

(A column 'c' in X is "trivial relative to b" if for all rows i, X[c][i]
can be determined from X[b][i], e.g the elements in X[c] are in
many-to-any correspondence with the values in X[b].)

The function will raise an exception if the list of pairs of values
in X[[a,b]] is not the product of the individual columns values, e.g.
        X[[a,b]]   ==   set(X[a])   x   set(X[b])   ,
in some ordering.
```

nullvalue(*test*)

Returns a null value for each of various kinds of test values.

2.2 Class DotData

Operations.DotDataFunctions.datahstack.numpy.core.records.recarray

Classes.DotData.DotData

A numpy recarray (a table with named columns where each column is of a uniform Python type), with added functionality and ability to define named groups of columns.

Invariants:

The names of all columns are distinct (unique) within one DotData.

2.2.1 Methods

__new__(*subtype*, *Array*=None, *Records*=None, *Columns*=None, *shape*=None, *dtype*=None, *formats*=None, *names*=None, *ToLoad*=None, *coloring*=None, *Path*='', *PathList*='', *SVPath*='', *SVDelimiter*=None, *SVDelimiterRegExp*=None, *SVLineBreak*=None, *SVHeader*=True, *SVHash*=None, *SVLineFixer*=None, *SVValueFixer*=None, *Wrap*=None, *SVSkipFirstLines*=0, *rowdata*=None)

```
Construct a new DotData from a copy of the data that's passed in.
     The new DotData is a separate object from all parts of the original data.

     There are several ways of creating a DotData.  You need to specify
     the table data, and the column headings.

     Specifying the data:

        - from a numpy.ndarray (Array arg)
        - from list of records (Records arg), where each record is represented
             as a sequence of elements
        - from list of columns (Columns arg), by passing each column as
             list of uniform Python values.
        - from a .data directory (Path arg);
        - from an SV (separated-values) file (SVPath, SVDelimiter,
             SVDelimiterRegExp, SVLineBreak, SVHeader, SVHash,
                     SVSkipFirstLines, SVLineFixer, SVValueFixer args)

     Specifying the column names:

       Column names can be inferred from the input data:
          - if you're constructing from a numpy.ndarray, the column names
             are taken from its dtype attribute
          - if you're constructing from .data directory, the column names
             are taken from the names of .csv
          files in that directory.
          - if you're constructing from an SV file, the column names
             can be taken from the headers in the file
          (by default they're assumed to be there).

       Or you can specify the column names as arguments:

          - you can specify column names as the 'names' argument
             (list of strings), or as the 'dtype' argument (numpy.dtype object).

       If you specify your own column names, and you're constructing from a numpy.recarray,
       they will override any column names from the numpy.recarray.

     You can specify a _subset_ of the columns to use, by specifying
     the list of column names as the ToLoad argument.

     Specifying the colors (named column groups):

Colorings can be inferred from the input data:

       If constructing from a .data directory, colorings will be automaticallly
       inferred from the directory tree.

Colorings can be passed as argument:
```

extract(*self*)

Creates a copy of this DotData in the form of a numpy.ndarray.

Useful if you want to do math on array elements, e.g. if you have a subset of the columns that are all numerical, you can construct a numerical matrix and do matrix operations.

selectRowsUsingOneCriterion(*self*, *expr*)

Return selected rows.

If expr is a string or a code object, returns rows for which expr is True. Expr is either a string or a code object; if a string, it is compiled into a code object. The expression is evaluated for each row, in an environment in which column names are bound to column values in that row. A DotData containing copies of rows for which the expression evaluates to True is returned.

If expr is a map of column names to values, returns rows for which the specified columns have the specified values. Column names mapped to None do not participate in the filtering.

You can get the same effect by using self[expr], but the [] operator is getting very overloaded...

See also: selectRows(), which can select rows according to a list of criteria.

selectRows(*self*, *expr*, **exprs*)

Returns a copy of rows matching all criteria in a list. See selectRowsAux() for documentation of the criteria.

This method takes a variable number of arguments; each argument is a separate criterion, and rows are returned which match ALL the criteria.

See also: __getitem__().

__**getitem**__(*self*, *attribute*)

```
Returns a subrectangle of the table.

The representation of the subrectangle depends on type(attribute). Also,
whether the returned object represents a new independent copy of the
subrectangle, or a "view" into this self object, depends on type(attribute).

- if you pass the name of an existing coloring, you get a DotData consisting of
copies of columns in that coloring

- if you pass a list of existing coloring names and/or column names, you get a
DotData consisting of copies of columns in the list (name of coloring is
equivalent to list of names of columns in that coloring; duplicate columns are
deleted).

- if you pass a Python expression as a code object (see compile_expr()), where
columns names are used in the expression as variables, you get subset of rows
for which the expression is true; more specifically, you get a DotData consisting
of copies of rows for which the expression evaluates to True, in an environment
where column names are bound to the corresponding values for each row.

- if you pass a dictionary mapping column names to values, you get back rows
where the specified columns have the specified values (more specifically, you
get a new DotData containing copies of these rows). Dictionary entries mapping
to None are ignored.

- if you pass an ndarray, you get a DotData consisting a subrectangle of the
table, as handled by recarray's: * if you pass a 1D ndarray of booleans of
len(DotData), the rectangle contains copies of the rows for which the
corresponding entry is True. * if you pass a list of row numbers, you get a
DotData containing copies of these rows.


See also: selectRows(), which lets you specify several row-selection criteria at
once, and lets you pass an expression for selecting the rows as a string rather
than only as a Python code object.
```

__**getslice**__(*self*, *attribute1*, *attribute2*)

Returns a slice into the array: a _contiguous_ range of its rows. This is not a copy but a mutable reference into the array.

vstack(*self*, *new*)

Create a new DotData that has rows of self followed by rows of new, with the headers, colorings and rowdata correctly merged.

hstack(*self*, *new*)

Create a new DotData that has columns of self followed by columns of new, with the headers and colorings correctly merged. rowdata is merged by simple SafeColumnStack (if there's repeated column names from different arrays, no exception is raised and the data from the first array that has that rowdata is taken)

addrecords(*self*, *new*)

Append one or more records to the end of the array. Can take a single record or tuple, or a list of records/tuples.

copy(*self*)

recordsAsDicts(*self*)

Return an interator over the records, which yields each record as a dictionary mapping column name to the value of that column in that record.

view(*self*, *obj*=`numpy.core.records.recarray`)

save(*self*, *TargetFolderName*, *HeaderOn*=`1`)

Save the DotData to a folder, preserving the colorings. Can later be loaded back by passing the folder as the Path argument to DotData constructor.

AppendToFile(*self*, *Target*)

Like "save" but for appending instead of writing from scratch.

ColIdx(*self*, *colName*)

Return the index of the column with the given name

mergeOnKeyCols(*dotDatas*, *primaryKeyCols*, *blanks*, *suffixes*, *verbose*=`False`)

Merge several dotDatas based on primary key columns.

Parameters:

dotDatas - sequence of DotDatas to join primaryKeyCols - sequence of column names, one for each DotData in dotDatas. these columns should all have the same type of data, typically an identifier telling us for what entity (e.g. for what SNP) that row gives information.

If each item in the sequence is a tuple of column names rather than a single column name, then the primary key from the corresponding DotData is the tuple of values from these columns, rather than a single value from one column.

blanks - when a DotData does not have a value for a given primary key, this tuple is filled in. suffixes - sequence of suffixes, one for each DotData, to append to names of columns from that DotData to make them unique, if needed.

For each primary key value existing in at least one DotData, we create one output record in the resulting merged DotData; these records are in order of primary key. The columns in the output are obtained by concatenating column name lists of the input DotDatas; any column names shared between two or more dotDatas are made unique by appending the corresponding suffix from 'suffixes'. The record corresponding to a given primary key is obtained by concatenating the records of the input dotDatas; when some input dotData does not have a row with the given primary key, the corresponding tuple from 'blanks' is used instead.

numCols(*self*)

Return the number of columns in this DotData

numRows(*self*)

Return the number of rows in this DotData

saveToSV(*self*, *TargetFileName*, *sep*=`None`, *linesep*=`'\n'`, *UseHeader*=`True`)

Save the DotData to a single flat file. Column headers are kept, but colorings are lost.

saveColumns(*self*, *TargetFolderName*)

Save the DotData to a set of flat .csv files in .data format (i.e. .int.csv, .float.csv, .str.csv). Colorings are lost.

isPrimaryKey(*self*, *columnName*)

Test whether the given column is a primary key, i.e. that each row has a unique value in this column, different from the value in any other row.

aggregate(*self*, *On*, *AggFuncDict*=`None`)

Aggregate a dataset, on specifed columns, using specified aggregation functions. See commends for DotDataAggregate function.

pivot(*self*, *a*, *b*, *Keep*=None)

Pivot a DotData table on columns a,b. See comments for Pivot function, and http://en.wikipedia.org/wiki/Pivot_table.

aggregate_in(*self*, *On*, *AggFuncDict*=None, *interspersed*=True)

```
Take aggregate of data set on specified columns, then add the
resulting rows back into data set to make a composite object
containing both original non-aggregate data rows as well as the
aggregate rows.

First read commends for aggregate method.  Now:
ARGUMENTS:
--On, AggFuncDict -- same as arguments for aggregate method.
-- interspersed : boolean, if true aggregate rows are interleaved
with the data of which they are aggregates, if false, all aggregate
rows placed at the end of the array.

RETURNS:
A DotData, with number of rows equaling:
                len(self) + len(A)
where A is the the result of self.aggregate(On,AggFuncDict).
A represents the aggregate rows and the other
rows were the original data rows.

This function supports _multiple_ aggregation, meaning that one
can first aggregate on one set of factors, then repeat
aggregation on the result for another set of factors, without the
results of the first aggregation interfering the second.  To achieve
this, the method adds (or augments, if already present), some
.rowdata information.  (See comments on __new__ of DotData
for more about rowdata information in general).
The specific rowdata information added by the aggregate_in
method is a column called "Aggregates" specifying on which
factors the rows that are aggregate rows were aggregated.
Rows added by aggregating on factor 'A' (a column in the original
data set) will have 'A' in the 'Aggregates' column of the self.rowdata array.
When multiple factors 'A1', 'A2' , ... are aggregated on, the notation
is a comma-separated list 'A1,A2,...'.    This way, when you call
.aggregate_in again, the function only aggregates on the columns
that have the empty char '' in their "Aggregates" rowdata.

The function also adds (or adds rows to) a '__color__' column
to self.rowdata, specifying Gray-Scale colors for aggregated
rows that will be used by the Data Environment system's
browser for colorizing the  data.   When there are multiple
levels of aggregation, the coarser aggregate groups
(e.g. on fewer factors) get darker gray color then those on
finer aggregate groups (e.g. more factors).
```

3 Package Operations.DotDataFunctions

3.1 Modules

- **AppendDotData** *(Section 4, p. 14)*
- **DictionaryOps** *(Section 5, p. 15)*
- **DotDataListFromDirectory** *(Section 6, p. 16)*
- **DotDataListFromSV** *(Section 7, p. 17)*
- **InferColoring** *(Section 8, p. 18)*
- **SaveColumnsInDotData** *(Section 9, p. 19)*
- **SaveDotData** *(Section 10, p. 20)*
- **SaveDotDataAsSV** *(Section 11, p. 21)*
- **datahstack** *(Section 12, p. 22)*
- **datavstack**: "Vertical stacking" of DotDatas, e.g. *(Section 13, p. 23)*

4 Module Operations.DotDataFunctions.AppendDotData

4.1 Functions

AppendDotData(*Data*, *TargetFolderName*)

Old deprecated append function for dotdatas.

AppendDotDataRecords(*path*, *RecObj*, *order*)

```
Function for appending records to an on-disk dotdata,
used when one wants to write a large dotdata that is
not going to be kept in memory at once.

If the dotdata is not there already, the function intializes
the dotdata using the regular DotData __new__ method,
and saves it out.

ARGUMENTS:
        path -- path of dotdata to append to
        RecObj : dictionary where:
                        -- keys are names of columns to append to
                        -- value on a column is a list of values to be appended to that column
        order -- ordering in which the columns should be written
                        -- only used when the dot-data does not exist
        and the header specifying order needs to be written

assumes a flat DotData structure (e.g. no colors)
        -- this should be generalized
```

5 Module Operations.DotDataFunctions.DictionaryOps

5.1 Functions

MergeDicts(**dicts*)

Construct a merged dictionary from the given dicts. If two dicts define the same key, the key from the dict later in the list is chosen.

RestrictDict(*aDict*, *restrictSet*)

Return a dict which has the mappings from the original dict only for keys in the given set

6 Module Operations.DotDataFunctions.DotDataListFromDirectory

6.1 Functions

DotDataListFromDirectory(*path*, *data_list*=None, *attribute_names*=None, *rootpath*='', *rootheader*=None, *coloring*=None, *ToLoad*=None, *Nrecords*=None)

```
Load DotData from a .data/ directory.
Each column is stored in a separate file
        (for column named myColumn the file is myColumn.datatype.csv
        where datatype is the data type of the column data ( int, int64, float, str, ... ),
        and the ordered list of columns is in a separate header file.)

See also: SaveDotData()
```

7 Module Operations.DotDataFunctions.DotDataListFromSV

7.1 Functions

DotDataListFromSV(*Path*, *Delimiter*=None, *DelimiterRegExp*=None, *LineBreak*=None, *Header*=True, *SVHash*=None, *ToLoad*=None, *SkipFirstLines*=0, *ValueFixer*=None, *LineFixer*=None)

```
Takes a tabular file with the specified delimeter and end-of-line character,
and returns a list of columns (i.e. each list item is a list of data in one column,
converted to native Python types) and a list of column names (column headers).

These two are returned as [attributes_list, attribute_names].

Parameters:

    Path - filename
    Delimiter - if given, the column separator in the file (if not given, automatically inferred fi
    LineBreak - if given, the line separator in the file (defaults to newline)
    Header - whether the first line of the file is a header line
    SVHash - whether we expect one or more hash-lines (comment lines starting
    with a hash) at top of file.
       If both Header and SVHash are True, then we assume the header line also
       begins with a hash,
       and is the last line beginning with a hash.
    ToLoad - list of attribute names (column names) to load.   each name in this
    list must match the name
       of a column in the file (and you must also have Header == True )
    SkipFirstLines - this many first lines of the file will be ignored
```

8 Module Operations.DotDataFunctions.InferColoring

8.1 Functions

InferColoring(*path*, *rootpath*=’’, *coloring*=`None`)

9 Module Operations.DotDataFunctions.SaveColumnsInDotData

9.1 Functions

SaveColumnsInDotData(*Data*, *TargetFolderName*)

10 Module Operations.DotDataFunctions.SaveDotData

10.1 Functions

SaveDotData(*Data*, *TargetFolderName*, *HeaderOn*=1)

11 Module Operations.DotDataFunctions.SaveDotDataAsSV

11.1 Functions

SaveDotDataAsSV(*Data*, *TargetFileName*, *sep*=`None`, *linesep*=`None`, *UseHeader*=`True`)

12 Module Operations.DotDataFunctions.datahstack

12.1 Functions

datahstack(*ListOfDatas*)

For "horizontal stacking" DotDatas, e.g. adding columns. Requires all DotDatas to have same number of records. If column appears to two DotDatas in ListOfDats, first is taken.

SafeColumnStack(*seq*)

13 Module Operations.DotDataFunctions.datavstack

"Vertical stacking" of DotDatas, e.g. adding rows.

13.1 Functions

datavstack(*ListOfDatas*)

SafeSimpleStack(*seq*)

Vertically stack sequences numpy record arrays. Avoids some of the problems of numpy.v_stack

nullvalue(*format*)

14 Package System

14.1 Modules

- **Colors** *(Section 15, p. 25)*
- **Gmail**: Contains functions for sending emails via gmail api.
  (Section 16, p. 26)
- **LinkManagement**: Functions that are used to extract and explore data dependency links.
  (Section 17, p. 27)
- **MetaData**: Routines for obtaining, generating, and processing meta data about data files, file runs, and functions used in the data file generation process.
  (Section 18, p. 38)
- **Protocols**: Functions for implementing data protocols.
  (Section 19, p. 40)
- **SVNOperations**: Routines for running the "flat" SVN archive that allows storage of files w/o .svn directories.
  (Section 20, p. 43)
- **Scriptification** *(Section 21, p. 45)*
- **StaticAnalysis**: Static analysis routines.
  (Section 22, p. 46)
- **Storage**: ========== Routines for storing and retrieving information about files and python objects.
  (Section 23, p. 48)
- **SystemGraphOperations**: Rountines for constructing meta-data graphs describing the link structure local to a give path in the Data Enviroment.
  (Section 24, p. 56)
- **Update**: Functions for automatic updating of data in the Data Environment by calling scripts indicated by data dependency links.
  (Section 25, p. 62)
- **Utils**: Miscellaneous utilities.
  (Section 26, p. 67)
- **Web** *(Section 27, p. 79)*
  - **DotData2Html** *(Section 28, p. 80)*
  - **HTMLCreators**: Convenience functions for html templates
    (Section 29, p. 81)
  - **Tabular2Html** *(Section 30, p. 82)*
  - **Tabular2HtmlOld** *(Section 31, p. 83)*
  - **WebRepresentations**: Unified web representation caller.
    (Section 32, p. 84)
  - **py2html**: create html representation of python module
    (Section 33, p. 85)
  - **text2html** *(Section 34, p. 86)*
- **config** *(Section 35, p. 87)*
  - **SetupFunctions**: Contains functions I use for configuring the Data Environment.
    (Section 36, p. 88)
- **extraction** *(Section 37, p. 89)*
- **system_io_override**: Intercept system i/o functions and analyze stack that made i/o calls.
  (Section 38, p. 91)

15 Module System.Colors

15.1 Functions

GrayScale(t)

Point2HexColor(a, *lfrac*, *tfrac*)

hsvToRGB(h, s, v)

Convert HSV color space to RGB color space

Parameters

`h:` Hue

`s:` Saturation

`v:` Value return (r, g, b)

16 Module System.Gmail

Contains functions for sending emails via gmail api.

16.1 Functions

Gmail(*account*, *password*, *to*, *subject*='', *Text*=None, *FileName*=None)

```
Sends an email from specified Gmail account to specified recipient address,
with subject and body from specified text or from text in specified file.

Account = string :  Gmail account name, e.g. if your gmail address was
"dyamins@gmail.com",
        then account would be set to "dyamins"
password = string: password for the above specified account.

To = email address to which the email will be send, e.g.
"info@barackobama.com"

Subject = string to be used as subject field of email, e.g. "Let's meet
Thursday"

Text = string : text to be used as body of email

FileName  = pathname string : if Text not specified, the contents of the file
FileName are read in and used as the body.
--> Either Text or FileName must be specified (if both, Text is used
preferentially.)

returns : nothing.  (but prints a status message after message is sent or
fails.)
```

17 Module System.LinkManagement

```
Functions that are used to extract and explore data dependency links.

The basic idea is that:
-- Some of these functions _FIND_ user-made python modules
(e.g. the scripts written by the user to make data computations
-- Some of these functions __ANALYZE__ user-made python modules
        to determine what data dependency links they contain
-- Some of these functions allow you to __EXPLORE__
        the dependency network described by these links
```

17.1 Functions

LinksFromOperations(*FileList*, *Aliases*=None, *AddImplied*=False, *AddDummies*=False, *FilterInternal*=True, *depends_on*=('../',), *creates*=('../System/StoredLinks/',), *Recompute*=False, *Parallel*=True)

```
Analyzes a set of python modules to find data dependency links present in the
modules.   Overall, this function does two things:

1) it writes out files that cache the links computed, and uses these cached
        versions to increase speed when the function is called again;
and

2) it returns the list of links, as a numpy record array.

Arguments:
--FileList = list of python modules to analyze, as a list of path strings.
--AddImplied = Boolean, which if True, adds implied links to the numpy array
        that is returned
--AddDummies = Boolean, which if True, adds Dummy links to the numpy record
        array that is returned
--FilterInteral = Filter returned list of links so that links between
        files and modules that are not in FileList are removed.
--Recompute = Boolean which if True causes the system to ignore
        cached LinkLists and recompute everything from scratch.
--Parallel = The algorithm includes the ability to parallelize if the system is
        being computed on has multiple processors.  If this boolean is True
        (the default) parallel computation is done.

Returns:
        A
where A is a numpy array describing the LinkList
```

GetImpliedLinks(*NewLinks*, *LinkList*)

This computes the implied links — assuming that all implied links in the "LinkList" argument have already by computed, it only computes the implied links that will be added by the addition of the NewLinks.

SameRoot(*X*, *Y*)

GetDummyLinks(*NewLinks*, *LinkList*)

This computes the dummy links — assuming that all dummy links in the "LinkList" argument have already by computed, it only computes the dummy links that will be added by the addition of the NewLinks.

GutsComputeLinks(*FileList*)

```
This function actually computes the links contained
in the files in FileList

Argument:
FileList == list of modules to analysis

Returns:
LinkList == list of links computed
Succeeded List = list of modules in FileList whose links
        could be successfully obtained.
```

ParallelComputeLinksFromOperations(*FileList*, *Parallel*=`True`, *creates*=`('../',)`)

Parallel wrapper for GutsComputeLinks

GetStoredDefaultVal(*op*, *name*, *NoVal*=None)

```
Given a python fnction in the form of a "Stored Operation",
return the function's default value for a given variable name,
returning NoVal if the variable doesn't exist for the function or doesn't
have a default value.

E.g. if the function is
                def F(x, y=3): ...
Then if op is the "Stored Operation" form of F,
        GetStoredDefaultVal(op,y) returns '3' ,
while
        GetStoredDefaultVal(op,x,[]) returns '[]'.

ARGUMENTS:
op -- a "Stored operation" -- Suppose you have a python function F.
Then the "Stored Operation" version of the function is a dictionary with:
key 'func_code' -- so that
        op['func_code'] = F.func_code,
the python code object associated with F, and key 'func_defaults'
-- so that
        op['func_defaults'] = F.func_defaults,
the tuple of default values of F.

name -- name of variable that you want to determine default value if

NoVal -- object to substitute in if the named variable does not exist for F
or if it that variable does not have a default value

Returns
        The default value when it exists, otherwise NoVal
```

ReduceListOfSetsOfScripts(*ScriptList*)

Given a list of sets of scripts, produces a reduced version retaining only the _last_ call to any given script.

GetLinksBelow(*Seed*, *Exceptions*=`None`, *Forced*=`False`, *Simple*=`False`, *Pruning*=`True`, *ProtectComputed*=`False`, *depends_on*=`('../',)`)

Given a Seed list of paths, loads the LinkList of live modules, and determines dowstream data dependency links by propagating downstream through the LinkList, from paths that are in Seed.

ARGUMENTS: Seed = list of paths from which to propagate downstream.

Forced = Boolean : if true, propagates links through LinkList without regard to timestamps of files, catching _all_ downstream links, even if the files they represent are up to date. If Forced is False – which is the default – this progates down only through links that are in need of updating, as determined by time stamps. (e.g. those link targets whose timestamps are old than their sources – and all the downstream files – are included.)

Simple = Boolean : if True, only looks at paths listed explicitly in Seed; if False, looks at all paths in Seed, as well as those in directories contained in the filesystem below paths in Seed

ProtectComputed = Boolean : if True, progates through links where target data files of links have been modified _after_ they were last created by system update of the link, e.g. as if the data had been corrupted after computation. Such files have a newer timestamp than their sources, so merely propagating from newer sources to older targets will not capture these files where computations (may) have been corrupted. Often times, one wishes to make a temporary change in a mid-stream file and see what its consequnces are, so in this case the ProtectComputed option should be left at its default value of False.

Exceptions = List : Before propagating downstream through the list, this function first filters out links whose scripts that are not indicated for Automatic updating (see below for comments in the function body). Exceptions is a list of scripts whose links should be retained even if they are not indicated for Automatic updating – the point of this is mostly for the function MakeUpdated (see comments in that function).

RETURNS: A = [L0,L1,L2 ...., LF] where A[i] = Li is sublist of links in the LinkList, representing those links activated at stage i in the downstream propagation through the LinkList from the seed links in L0.

In words, this is a trace through the Directred Acyclic Graph represented by the LinkList. Note that the same link can appear multiple times in different stages because it might be activated first by itself (e.g. because its source is in Seed and its target is older than its source) and then by virtue of downstream propagation from some independently activated link upstream.

GetII(*LinkList*, *Seed*)

UpdateGuts(*UpdateList*, *LinkList*, *MtimesDict*, *PtimesDict*, *ProtectComputed*)

```
The guts of the downstream propagation through a linklist.
Used primarily by the function PropagateThroughLinkGraphWithTimes.

ARGUMENTS:
LinkList = A numpy record array of data dependency links.

UpdateList = List of the form [(i1,t1),(i2,t2), ... , (in,tn)] where
each  i is an index in LinkList, and ti is a time stamp (or a 'numpy.nan')

MtimesDict = Dictionary where:
-- the keys are some of the elements of LinkList['LinkSource']
        or LinkList['LinkTarget']
   These are paths in filesystem, or subparts of paths. (e.g. functions inside .py
   module files)
-- the values values on key LinkList['LinkSource'][i] is

FindMtime(Linklist['SourceFile'][i],functionname=LinkList['LinkSource'][i])

these are the timestamps associated with a path or a subpart of a path

ProtectComputed = Boolean : if true, enables propagation along links
whose targets have been modified after creation

PtimesDict = Dctionary used when ProtectComputed = True, where:
-- the keys are paths in filesystem
-- the values on a key P is the most recent time that the file or
directory at time P was successfully computed by the automatic
updating facility

RETURNS:
TargetRecs a numpy record array of containing list of links in
LinkList that are activated, relative to the information provided
in the UpdateList.  Essentially these are:

-- all those links (source,target) in LinkList[i[0] for i in UpdateList]
such that MtimesDict[target] is less than then T, where T is maximum of:
        MtimesDict[source]
and
        the maximum of

        [i[1] for i such that LinkList['LinkSource'][i[0]] == source]

-- plus those that are activated when ProtecteComputed is True

Some Information is appended to each link that is activated,
including the "maximal propagation time" along the link -- which is:
        -- T, in the case of a "Uses" link
        -- numpy.nan, in the case of "CreatedBy" link
```

PropagateThroughLinkGraphWithTimes(*Seed*, *LinkList*, *Simple*=`False`, *Pruning*=`True`, *HoldTimes*=`None`, *ProtectComputed*=`False`, *depends_on*=`('../',)`, *creates*=`('../System/MostRecentTrace.dot',)`)

```
Given a Seed list of paths, and Linklist propagates downstream
through the LinkList from the Seeds, taking into account
timestamps along the way.

ARGUMENTS:
Seed = List of paths to start with.

LinkList = numpy record array of data dependency Links.

Simple = Boolean : if True, only looks at paths listed explicitly in Seed;
if False, looks at all paths in Seed, as well as those in directories
contained in the filesystem below  paths in Seed

ProtectComputed = Boolean : if True, progates through links
where target data files of links have been modified _after_ they
were last created by system update of the link, e.g. as if the data
had been corrupted after computation.

HoldTimes = Dictionary, where:
-- keys are paths
-- HoldTimes[Path] is a timestamp that the system is meant
to "pretend" is the mod time of Path, if Path comes up as a
source or target during the propagation process,
instead of computing the real mod time.

RETURNS:
        A = [L0,L1,L2 ...., LF]
where A[i]  = Li is sublist of links in the LinkList, representing
those links activated at stage i in the downstream propagation
through the LinkList from the seed links in L0.

In words, this is a trace through the Directred Acyclic Graph
represented by the LinkList.   Note that the same link can appear
multiple times in different stages because it might be activated
first by itself (e.g. because its source is in Seed and its target is
older than its source) and then by virtue of downstream propagation
from some independently activated link upstream.
```

PropagateThroughLinkGraph(*Seed*, *LinkList*, *depends_on*=('../',))

```
Given a Seed list of paths, and Linklist propagates downstream
through the LinkList from the Seeds.

ARGUMENTS:
--Seed = List of paths to start with.
--LinkList = numpy record array of data dependency Links.

RETURNS:
        A = [L0,L1,L2 ...., LF]
where A[i]  = Li is sublist of links in the LinkList,
representing those links activated at stage i in the downstream
propagation through the LinkList from the seed links in L0.

In words, this is a trace through the Directred Acyclic Graph
represented by the LinkList.   Note that the same link can appear
multiple times in different stages because it might be activated
first by itself (e.g. because its source is in Seed and its target is
older than its source) and then by virtue of downstream
propagation from some independently activated link upstream.
```

PropagateUpThroughLinkGraph(*Seed*, *LinkList*, *depends_on*=('../',))

Like PropagateThroughLinkGraph, but goes upstream from Seed instead of downstream.

PropagateSeed(*Seed*, *LinkList*, *N1*, *N2*, *N3*, *Special*)

```
Does the maing propagation of a seed through a linklist.
A fairly general graph propagation function.  In words,
this is a (basically) a trace through the Directred Acyclic Graph
represented by interpreting (LinkList[N1],LinkList[N2]) as the
edges of a graph -- together with the implied paths by path inclusion.

ARGUMENTS:
--Seed = List of paths to start with.
--LinkList = numpy record array of data dependency Links.
--N1 = Column header in LinkList ; Representing the "source" column
--N2 = Column header in LinkList ; Representing the "target" column

RETURNS:
        A = [L0,L1,L2 ...., LF]
where A[i]  = Li is sublist of links in the LinkList,
representing those links activated at stage i in the propagation along
direction N1 to N2 through the LinkList from the seed links in L0.

Note that the same link can appear multiple times in
different stages because it might be activated first by itself
(e.g. because its source is in Seed and its target is older than its source)
and then by virtue of downstream propagation from some
independently activated link upstream.

If the graph ends up being cyclic, it prints an error message and
returns an empty recarray with the same fields as LinkList.
```

GetI(*List*, *Seed*)

GetConnected(*Seed*, *level*=`-1`, *Filter*=`True`, *depends_on*=`('../',)`)

```
Convenience function printing out targets at specified level away from seed in
link depedency network, for the LinkList loaded from live modules.

ARGUMENTS:
--Seed = List of paths to propagate away from.
--level = integer : level away from seed to find ;
        -K means K leves upstream, +K means K levels downstream.

RETURNS:
--Python set object containing paths of dependencies, if any.
(If none a message is printed.)
```

LoadLiveModules(*LiveModuleFilters*=`None`)

Makes a a list of live modules in the Data Environment filesystem, to inspect for links. Determined by reading contents from user specified configuration file. (see comments in body of the code)

OVERALL: The strategy is going to be to get a list of regular expression filters to apply a raw candidate list of directories; then recursively list files in the directories that survive these filters; and then return the .py files among this list. The way that the files are recursively searched can be done by user a user-specified search function, a python function in a module whose python 'dot path' is specified by setting the environment variable 'DotPathToLiveModuleFilterFunction.' If this is not a specified a default version is taken.

First, getting the list of filters: Either it's given (whence it's not None); or if it is None, then parse a user-specified configuration file that lists filters. The user-specified file's path is given by an environment variable called 'LiveModuleFilterPath'. The format of the filters is: a pair (D,F) where D is a directory to search in, and F = (r1,r2,....,NOT rn) is a list of regular expressions or regular expressions preceeded by 'NOT ' The format of the user-specified file is: a list of lines, each of which species a (D,F) pair in the following way:

Path1 : r1, r2, NOT r3, r4, Path2 : r1, NOT r2 Path3

Files from Path1 survive if the match any of the regular expressions r1, r2, r4, ... but NONE of the expressions preceeded by a 'NOT ', e.g. r3. If the line contains no ':' but only contains a directory path, than the regular expression is assumed to be allow everything in the directory.

White space before and after the ':' in each line is ignored, as is white space before and after each comma separating the regular expression.

For instance the follow is valid:

../Operations/: ^../Operations/[^/]*$, ^../Operations/Dan_Operations/* , NOT

^../Operations/Dan_Operations/ExamplesOfPCA*,NOT
^../Operations/Dan_Operations/Dan_SystemGraph*

../Users/DanYamins/

ARGUMENTS: –An optional list of Filters can be passed to this function, though it is uncommonly used.

RETURNS: –A python list containing paths of python modules.

FilterForAutomaticUpdates(*LList*, *Exceptions*=None, *AutomaticUpdateFilters*=None, *ReturnIndices*=False)

```
Filters a link links removing links that are not meant to be
automatically updated when the system runs downstream
updating -- except allows "exceptions" to pass.
The strategy is to basically get the list of filters from a
user-specified configuration file whoe path is given by the
environment variable 'AutomaticUpdatesPath'.

ARGUMENTS:
--LList = LinkList as a numpy record array from which to filter.
--Excetions = List of paths of scripts that should not be filtered out, regardless.
--AutomaticUpdateFilters  : an optional list of filters can be passed in.  (but usually isn't)
--ReturnIndices:  boolean which if true means that the function will return
the _indices_ in LinkList that are to be retained instead of the retained
links themselves.

RETURNS:
If ReturnIndices = False:
        A subarray of LList if
else:
        A numpy array of indices of LList.
```

DownstreamFiles(*Sources*, *depends_on*=('../',), *Filtering*=False)

A convenience function listing file downstream from a specified source list in the LinkList dependencies in live modules.

ARGUMENTS: –Sources: list of sources to propagate downstream from. –Filtering: boolean, which if set true retains only those targets that will be automatically updated.

DownstreamAlongLinks(*Sources*, *LinkList*, *depends_on*=('../',))

UpstreamFiles(*Targets*, *depends_on*=('../',))

Analogous to DownstreamFiles.

UpstreamAlongLinks(*Targets*, *LinkList*, *depends_on*=('../',))

UpstreamLinks(*Targets*, *depends_on*=('../',))

17.2 Variables

Name	Description
isnan	**Value:** numpy.isnan
nan	**Value:** numpy.nan

continued on next page

Name	Description
DefaultValueForAutomaticUpdates	**Value:** `['^../(.*)']`
DefaultValueForLiveModuleFilters	**Value:** {`'../Operations/': ['../*']`}

18 Module System.MetaData

```
Routines for obtaining, generating, and processing meta data about data files,
file runs, and functions used in the data file generation process.

The way that the metadata scheme works is:

In the top level System Folder there is a directory ../System/MetaData that
contains a replica of the file structure outside of the System folder.
Each "real" path corresponds to a directory in the MetaData directory,
in which metadata about the path is stored,
e.g. '../Data/Dan_Data/NPR_Puzzle_Solutions', would correspond to
'../System/MetaData/Data/Dan_Data/NPR_Puzzle_Solutions'.
Metadata is similarly attached to functions in python modules referenced via their dot-paths.

Given a file path P, the associated path for the metadata is given by the
function metadatapath(P).  Given a python function dot path, the associated
path P for the metadata is given by the function opmetadatapath(P).

Inside a given path's metadata directory is a variety of associated metadata files.
There are three main kinds of associated things:
        -- data generated during the runtime that produced/was generated by the path.
        -- metadata attached by the human directly to the file.
        -- data used and generated by the system Graphical Browser.
        e.g. Linklists local to the path, and graphviz-generated graphs of these LinkLists.
        (See commends in System/SystemGraphOperations.py for information about this)
```

18.1 Functions

AttachMetaData(*NewMetaData*, *FileName*='', *OperationName*='', *creates*=('../System/MetaData/',))

```
Attach metadata to a file a given path.

ARGUMENTS:
--NewMetaData : a dictionary that will be pickled as metadata.
--FileName = Path of the file to which the metadata should be attached.
--OperationName = Python dot-path of the operati which the metadata should
be attached.
(Either File or OperationName should be specified, but not both.)

Metadata attached through this process is put into the file
        MetaDataPath/AttachedMetaData.pickle.
If this path exists at the time that an AttachMetaData command is run,
the metadata dictionary in the file is updated with the NewMetaData dictionary.
```

MakeRuntimeMetaData(*opname*, *Creates*, *OriginalTimes*, *OriginalDirInfo*, *RunOutput*, *ExitType*, *ExitStatus*, *Before*, *After*, *IsDifferent*)

```
This is an internal usage function that attaches the results of
metadata generated during the running of a system update.
It is used primilary by the UpdateLinks function in ../System/Update.py

Suppose the function F is run by the automatic updater.
During this runtime, several kinds of data are produced:
--the return output of the function F, and
--information about the run, its Exit status, whether it changed any
        files it was meant to produce, etc...

These pieces of runtime metadata are collected by the UpdateLink fnction
during update, and then passed to MakeRuntimeMetaData function to write
out the data in an appropriate format to the metadata files.

in summary, it
-- writes output of the executed function F to a file at path
        opmetadatapath(F) + '/RuntimeOutput.pickle'
-- appends exit status information to opmetadatapath(F) + '/ExitStatusRecord.csv'
-- for each file j created by the running of F:
        -- appends file-specific creation information to
        metadatapath(j) + '/CreationRecord.csv'
        -- attaches MetaData returned by the script for file j to
        metadatapath(j) + '/AssociatedMetaData.pickle'
```

IsFailure(*Path*)

Returns Boolean True if Path represents is the python dot-path of an operation whose most recent run by the autmatic updater was a failure.

GetBrokenOperations(*depends_on*=`('../System/MetaData/',)`)

Returns list of operations that failed on their most recent run.

LastTimeChanged(*path*)

Returns last time, according to runtime meta data, that a file (at "path") was actually modified (e.g. not simply overwritten, but actually modified.)

FindPtime(*target*, *Simple*=`False`)

Returns last time, according to runtime meta data, that a target was succesfully created, if it is created data.

metadatapath(*datapath*)

opmetadatapath(*oppath*)

19 Module System.Protocols

Functions for implementing data protocols.

19.1 Functions

ApplyOperations(*outfilename*, *OpThing*)

This is deprecated and only kept around to support a couple of initial protocols built using it.

ApplyOperations2(*outfilename*, *OpThing*, *WriteMetaData*=`True`)

```
Function which implements the basic protocol concept.
Given an 'OpThing', which is a set of descriptions of operations
and corresponding arguments to be passed to the operations,
this writes out a python module with functions making the calls.

ARGUMENTS:
outfilename = pathname for the output python module, must end with .py
OpThing = dictionary or list.
1) if dictionary, then:
-- the keys are strings representing names for the concrete operations in the
        written-out python file
-- for each key, say 'FName' the value is a 3-tuple: (function, internal arguments,  external argu
        where:
--function is the function -- the actual python function object, NOT the just the
        function name -- to be called as the body of FName

--internal arguments are values for the arguments to be pased to FName
The internal arguments can be given in one of several forms:
        --a tuple of positional arguments
        --a dictionary of keyword arguments
        --a 2-element list [t,d] where t is a tuple
        of positional arguments and d is a
        dictionary of keyword arguments

--external arguments is a dictionary where:
        the keys are names of keyword arguments for FName
        the values are the default values for the keyword arguments

2) if a list then:
        each element of the list is a 4-tuple

        (FName, function, internal arguments, external arguments)

        where all these are as described just above.
        The only difference between the list-frmat input and
        the dictionary format input is that the list format
        takes the keys of th dictionary and makes the
        first element of the tuples in the list
        (which are now 4-tuples, as opposed to 3-tuples in
        the dictionary case).

        The reason for the list format is that it the operations
        are written out to the file in the order specified in the list,
        whereas with a dictionary the order is  determined by the
        order of writing out the keys of the dictionary, which is
        often not the "natural" ordering.

WriteMetaData: boolean, which if true has the operations write
metadata for th operations in the outputted module.

RETURNS:
        NOTHING
```

OpListUniqify(*OpList*)

When the OpList in the ApplyOperations2 contains multiple functions that are the same, e.g. have the same contents, but perhaps not the same FName,this optimizes and retains only one of each.

20 Module System.SVNOperations

Routines for running the "flat" SVN archive that allows storage of files w/o .svn directories.

The basic idea is that: files that the user wants to put in SVN are copied, with a naming convention, to a single large flat SVNArchive directory. Then, files in this directory are actually the ones put under SVN control.

This module provides convenience functions for managing this process, including CheckIn(), Update(), and Status().

Each of these functions has 3 basic steps:

1) before doing anything, the archive is "Equalized", meaning that – at least for the directories the user has specified in a configuration file – the achive is made to contain the exact same contents as the user's real directories. The user's file system is always treated with priority – whatever the contents there, that is what the archive is made to reflect.

2) then the SVN action occurs, be it Check In or Update, or Status.

3) the the users' real file system is Equalized with the archive, to get any new or modified files or delete those that have been removed in the SVN repository.

To use these routines the environment variable 'SVNArchivePath' should be set – this reflects the location of the SVNArchive.

Each of them takes an optional SVNPath argument, which can be used in place of the users' specified confiuration file.

20.1 Functions

CheckIn(*SVNPaths*=`None`)

Run checkin on the the "flat" SVN archive from user specified directories.

ARGUMENT: –SVNPaths = list of directory paths to look in. Normally this argumnt is not given, and instead the files are generated by a user-specified configuration file. See commens for the function "ProcessSVNPaths"

Update(*SVNPaths*=`None`)

Run update on the the "flat" SVN archive from user specified directories.

ARGUMENT: –SVNPaths = list of directory paths to look in. Normally this argumnt is not given, and instead the files are generated by a user-specified configuration file. See commens for the function "ProcessSVNPaths"

Status(*SVNPaths*=`None`)

Run svn status on the the "flat" SVN archive from user specified directories.

ARGUMENT: – SVNPaths = list of directory paths to look in. Normally this argumnt is not given, and instead the files are generated by a user-specified configuration file. See comments for the function "ProcessSVNPaths"

EqualizeArchive(*Files*, *SVNPaths*, *ArchivePath*=`None`)

Equalizes archive to reflect exactly identical contents to the users' filesystem outside the archive.

EqualizeRealFiles(*Files*, *SVNPaths*, *ArchivePath*)

Equalizes users real filesystem to reflect contents of SVN archive.

ProcessSVNPaths(*SVNPaths*)

Given list of paths, returns a list of files in these paths to copy to and from the SVN archive.

ARGUMENT: – SVNPaths = a list of paths, or None

If SVNPaths = None, the function looks for an enviroment variable called PathToSVNInfo, which specifies the path to a user-specified file containing a list of directories to use as SVN paths. (One directory per line)

Once SVNPaths is determined, the files in those directories are determined. This can either be done by the default, which is simply to get recursively all files in teh directories, or by some user-specified function. To make the latter work, you specify the environment variable 'DotPathToSVNFilterFunction' which should be a python dot path to a python function. This function must take the argument SVNPaths (which may be None), and must return [Files,SVNPaths].

AddAll(*ArchivePath*=`None`)

calls a simple command line awk script that svn adds all non-added files in a directory

DeleteMissing(*ArchivePath*)

calls a simple command line awk script to svn del all missing fles in a directory USE WITH CARE!!!!

RealName(*apath*, *ArchivePath*=`None`)

converts an archive file path to the corresponding path that the 'real file' would have

SVNArchiveName(*path*, *ArchivePath*=`None`)

converts a 'real file" path to the corresponding path that copy in the SVN archive should have

20.2 Variables

Name	Description
DefaultSVNArchivePath	**Value:** `'../SVNArchiveArea/'`

21 Module System.Scriptification

21.1 Functions

CommandDependencyIndices(*VarNames*, *CommandHistory*)

UnrollGetatt(*getseq*)

GetNames(*AST*)

Scriptify(*VarNames*, *CommandList*, *ToFile*, *AsFunction*=`None`)

WriteToPyModule(*CommandLines*, *ToFile*, *AsFunction*=`None`)

MakeScript(*VarNames*, *ToFile*, *AsFunction*=`None`)

22 Module System.StaticAnalysis

Static analysis routines.

22.1 Functions

GetFullUses(*FilePath*)

```
Adds information about actual paths in  existence in the file system to further
refine results of the GetUses function.

Argument:
--FilePath = path to file of module to do static analysis of.

Returns:
If call to GetUses Fails, then: None, else, a dictionary F, where:
-- the keys are names of objects defined in the module in FilePath
and
-- for the key 'Obj', the value F['Obj']  is a list of pairs

        (objname,file)

        where objname is the name of an object that is referenced
        somwhere in the definition of 'Obj' and file is the name of the
        file where objname is defined.
```

GetUses(*AST*=None, *FilePath*=None)

```
Gets information about the names referenced in a python module or AST
fragment by doing static code analysis.

ARGUMENTS:
--AST = pre-compiler parse tree code object,
        e.g. result of calling compiler.parse() on some code fragment
--FilePath = Path to python module
One or the other but not both of these two arguments must be given

RETURNS:
None, if FilePath is given and call to compiler.parseFile() fails, else:
AST is either given or generated frm the FilePath code returns a 3-element
list [F,M,N] where:
---F = dictionary, where:
        the keys are names of scopes (e.g. classes and functions) defined in the AST
        the values are names of functions and names mentioned in the functions
--M = list of modules imported or used in the AST
--N = dictionary where
        -- the keys are names of non-scoping names (those besides functions or classes)
        defined in the AST
        -- the value associated with key 'X' is list of other names
        by which X is known in the AST scope, including its definition
        in terms of imported modules
```

GetUsesFromAST(*e*, *NameDefs*, *ModuleUsage*, *NamesUsage*, *CurScopeName*)

Recursive function which implements guts of static analysis.

ProperOrder(*NodeList*)

Order a list of AST nodes so that their analysis happens in proper for name defintions to be taken into account – e.g. nonscope nodes first, then scope nodes.

interpretation(*n*, *NameDefs*)

determine "real" name(s) of a getattr or name ast node in terms of names of its pieces

UnrollGetatt(*getseq*)

unrolls a Name or GetAttr compiler ast node into a dot-separated string name.

DictionaryOfListsAdd(*D*, *key*, *newitem*)

23 Module System.Storage

```
===========
Routines for storing and retrieving information about files and python objects.
===========

In the Data Environment, which is devoted to understanding and managing dependency links,
routines throughout the system often require stored information about files, directories,
functions, and data structures.   This information may include:

1) Information that can't directly be determined by analyzing the file or directory:
for example, the file's last modification-time or create-time.    This information is
often necessary to determine when an object has changed, so as to know when to
trigger the re-running of a process that depends on the object.   This is the case
with the function PropagateThroughLinkGraphWithTimes, in the LinkManagement
module, which needs to compare link target's mod times to link source's mod
times to know when to re-run a data-creation script.

or,

2) Information that is that _is_ possible to glean upon complex introspection but hard
to obtain otherwise: for example, the list of names of python functions that might be called
by another function during its run.  This information is useful to analyze an object to
determine what its dependencies are in the first place.   This is the case with the function
ComputeLinksFromOperations, which determines the data "depends_on" and "creates",
and functional "uses",  from a python function's code.

This module provides a unified interface to obtain such information so that  it can be
called up wherever needed.   The key realizaion is that providing unified access to the
relevant information about the objects boils down to having a unified method for
_storing_ and retrieving "stylized versions" of those objects,  versions that contain
somewhat detailed information about the objects' parts, as well as information about
when those parts were last modified.

This is because:

1) if you need to get information about when an object has changed, you'll need to have a
stored version that can be compared to the current version to detect chages, together
with timing information about each of those stored parts so that if the part _hasn't_
changed you can tell what the last actual mod-time of the part was.

and

2) to provide easy access to complex introspected information, it makes sense to
compute that introspection only once per object per modification, and store the
results of that introspection in a standardized format in a single place.

The  "data model" behind our approach to doing this enables us to be as lazy as possible,
leaving as much to operating system as we can.  The basic idea is that every object
stored on disk in the Data Environment file system is one of two things:
```

```
        1) A directory or a file inside a directory  -- whose properties are "at the operating system le
                and which don't need to be made "live" to access
or
        2) an implied object __inside__ a file -- whose properties are "more specific than
        the operating system level" and which require some form of "being live" to access

For instance, a python package consists of a directory containing python .py files, and
in turn each py file is a module containing python objects.  Or a relational database,
which is at one level a filesystem, but which at a lower level consists of records.

Now, for information about things at the files and directories levels, we are able to
rely mostly on things provided by the operating system: it obviously already stores
the files themselves, and provides access to file and directory modification
information, through things like the 'stat' and 'diff' utilities.     However, for finer
detail, we need to supplement the operating system.   The basic strategy is:
        -- for each of several "Special Supported File Types", store information about the
        more detailed parts of the supported file type
and then
        -- provide an _extension_ of the stat and diff utilities that allows queries to specify
        both a file path, as well as a more detailed  part-name.

For instance, the standard python implementation of the path mod-time function is
'getmtime' function in the the os.path module.  The function os.path.getmtime takes
as argument a single path, and returns the time of last modification of that path.
Here, we extend that function to the FindMtime function which takes both a pathname
argument, as well as an "object-name" argument, and which returns the mod-time of
that specific object.  For the moment, we only have one "Special Supported File Type":
python modules ; so the "objectname" argument boils down to a "function-name"
argument that allows one to find the mod-time for a given function in a module.
All other files are treated at the atomic level.  (In the future, other kinds of file parts
could be stored, e.g. records in a database by integrating database query routines .... etc...)
```

23.1 Functions

FindMtime(*path*, *objectname*=’’, *HoldTimes*=None, *Simple*=True,
depends_on=(’../System/StoredModules/’,),
creates=(’../System/StoredModules/’,))

```
This is the main unified interface to path and path-part mod times that
is to be used throughout the system.

ARGUMENTS:
--path = path whose mod-time is to be assessed
--objectname = name of object within that path, whose mod-time is to be assessed.
--HoldTimes = Dictionary, where:
        --keys are paths
        --HoldTimes[Path] is a timestamp that the system is meant to "pretend"
        is the mod time of Path, if Path comes up as a source or target during the
        propagation process, instead of computing the real mod time.
-- Simple = Boolean : if True, only looks just at the modtime of path; if False,
and if path is a directory, it looks recursively through the mod times of files
inside path and returns the maximum.

RETURNS:
-- floating point number representing a mod-time, in seconds since the start of the
Unix Epoch.  (Jan 1, 1970 at 00:00:00 GMT).

NB: For the moment, the object name parameter only does anything if the path
is a python module, and if then only if the objectname names an object defined
in that python module.   The way this is determined is by a combination of
"live analysis" -- e.g importing the python module and inspecting its contents
via introspection methods;  and "static analysis" which relies on analyzing
python code without importing the module but which does require building
the compiler parse tree for the code.  If either the importing step or the
compiler parse-tree step fails, the function just returns the mod-time of the
file that the module is in.  [All of this is sort of built haphazardly in to the
way the FindMtime function and its depedencies are written.  In the future,
to accommodate other "Special File Types", the way this and associatd
functions are written would be to be made more modular.]
```

ListFindMtimes(*FileParts*, *HoldTimes*=None, *Simple*=True, *Parallel*=True, *depends_on*=('../System/StoredModules/',), *creates*=('../System/StoredModules/',))

```
Given a list of files and objects within those files, computes mtimes for them.
This is an optimization on top of FindMtimes, by inspecting a list of file/object pairs,
then analyzing the uniquely mentioned files only once.

ARGUMENTS:
--FileParts: a python list of pairs (FileName, ObjectName) each of which is to
be analyzed for mod-time.   If only a FileName is meant to provided,
e.g. no subpart is to be looked for, just the mod-tiem fo the whole file,
than ObjectName should be set to equal FileName.

For instance to get the modtime of the function "GetBasketBallTeams"
in the file:

'../Users/Elaine/Playbox/Sports/ESPN_NBA/NBA_Teams.py'

you'd include in the list the pair:

('../Users/Elaine/Playbox/Sports/ESPN_NBA/NBA_Teams.py',
'Users.Elaine.Playbox.Sports.ESPN_NBA.NBA_Teams.GetBasketBallTeams')

but to get the modtime of the file:

'../Users/Elaine/Playbox/Sports/ESPN_NBA/NBATeamData.data',

you'd include the pair

('../Users/Elaine/Playbox/Sports/ESPN_NBA/NBATeamData.data', '../Users/Elaine/Playbox/Sports/ESPN_N

--HoldTimes, Simple are the same as the correpsonding FindMtimes arguments
--Parallel = boolean indicating whether the function should be executed in
parallel on multiple processors in a machine (if available)

Returns:
A dictionary, where:
        --the keys are the unique object names (which are same as the
        file names when no specific sub-object is to be given)
        --the value at a key is the same as would be there if
        FindMtims(FileName,ObjectName) were called.
```

BlockUpdateModuleStorage(*L*)

GetStoredModule(*path*)

Returns stored module for the python module whose file is at 'path'. If the module storage update process fails, this function returns None. The format of the returned object is a dictionary where each key is the name of the module parts, and the value at the key is the instance of the StoredModulePart class for that part. (see below in StoredModulePart for details)

GetStoredModuleTimes(*path*)

Returns stored module's mod times for the python module whose file is at 'path'. If the module storage update process fails, this function returns None. The format of the returned object is: a dictionary whose keys are the same as the keys of the stored module dctionary, and whose values on those keys are mod times for the parts. (There's an extra key called '__hash__' which stores a hash of the stored module dictionary to ensure data integrity upon update.)

GetNestedObject(*name*, *Members*)

Technical dependency of ExtractParts

GetExtendedMembers(*obj*, *Static*)

Technical dependency of ExtractParts

GetExtendedExecedNames(*Execed*, *Static*)

Technical dependency of ExtractParts

ExtractParts(*obj*, *Execed*=None, *Static*=None)

```
Given a python object obj, extract it for storage.   This is called by the function
UpdateModuleStorage and the class StoredModulePart.

ARGUMENTS:
--obj = the python object to get stored parts of --- as "made live" by having
        been imported in a module ultimately in UpdateModuleStorage.
--Execed = an _execfile'd__ version of the same object, as opposed to imported.
--Static = the static analysis version of the object.

Returns:
--Dictionary whose keys are the names of parts of the object and whose
values on each key are the stored versions of the part with that name.

NOTE:  The execfiled and static analysis objects supplement and
verify the information stored about the object.
```

UpdateModuleStorage(*path*)

```
Updates the file storing the module at path 'path', as well
as the file storing the mod-times for those parts.

ARGUMENT:
-- path = the path of the module whose storage is to be updated

RETURNS:
-- Nothing.  But it updates the module storage.   To get at the
stored objects you use one of the two functions
(GetStoredModule or GetStoredModuleTines)

The basic format of storage is:  given a module file, to associate
two storage files with it:
        1) a file containing a pickling of a StoredModule dictionary
        containing stored versions of the parts of the module,
        at path StoredModulePath, and
        2) a file containing a pickling of a StoredModuleTimes dictionary
        containing the Modtimes for each of the stored parts,
        at path StoredTimesPath.

The StoredModule dictionary's format is to associated to each part-name
an instance of the StoredModulePart class:
        StoredModule[ part-name ] =   StoredModulePart(part),
See comments in StoredModulePart class for details.

The StoredModuleTimes format is a dictionary whose keys are the same
as the keys of the StoredMdule dictionary, and whose values on those
keys are mod times for the parts.
(There's an extra key called '__hash__' which stores a hash of the stored
module dictionary to ensure data integrity upon update.)

The reason that the StoredTimes file is stored separately from the
StoredModules file, instead of in one big dictionary is that this way,
the much smaller StoredTimes dictionary can be loaded for use in
evaluating functions like FindMtime, without having to lead the
whole module storage.

This function basically has two stages:

1) First, determine whether the module's object storage is:
        -- Update to date, in which case nothing has to be done
        -- In OK format but may not be up to date, in which
                case the module nees to be imported and analyzed,
                the results compared to the stored version
        -- Non-existent or somehow contaminated or in the wrong format,
                in which case it needs to be remade from scratch.

2) Having determined which state the storage is in, and if the
module storage needs to be updated, act upon that.
The action consists of:
        - pickle-loading the StoredModule already on disk if it exists,
                (as well as the stored module mod times, which is already
                unpickled in stage 1 of this function)
        -- computing a new version of the stored module,
        -- for each part in the new Stored module, comparing it to the stored
```

ClassMethodDumps(*obj*)

Storage method for class methods.

FunctionDumps(*obj*)

Storage method for functions.

CodeEquals(*c1*, *c2*)

Equality testing for code objects.

GetStoredPathNames(*path*)

Determines path names from stored module and stored module times obects, from path name of the module to be stored.

23.2 Class StoredModulePart

This class defines the storage for a module. Calling StoredModulePart(object) creates a storable version of the live object.

If it were possible to pickle all kinds of python objects, this class would be unnecssary – we'd just use the Pickle.dumps method. However, it is not possible to run standard python pickling on all python objects.

This is for two reasons:

1) Some python objects require special pickling methods – e.g. code objects require "marshal" – because cPickle wasn't built to handle them (for good reason). But since we want to store many objects types together, we have to have a single interface for all the various kinds of storing.

2) The standard of "being picklable" means that the pickle.load method has to completely restore the unpickled object to fully functional form. This is basically impossible for things like functions or class methods or modules because they rely on defined names in the module context or from other modules, so no pickling method for them exists. But for our purposes we don't actually need to require so high a standard of our storage: we don't need to use the stored objects as live substitutes for the real objects, we merely need just to be able to recover enough detail to be able to determine whether the live real object differs in some material way from the stored one.

This StoredModulePart class basically does two things therefore: 1) unifies all the various pickling methods to do exist into a single interface, and 2) extracts data about the objects into a set of objects that are "dry" enough that they _can_ be pickled, but which are descriptive enough to provide the ability to make meaningful comparisons to check for modifications.

23.2.1 Methods

__init__(*self*, *obj*, *ScopeName*=`None`, *Static*=`None`)

reconstitute(*self*)

__**eq**__(*self*, *other*)

23.2.2 Class Variables

Name	Description
__module__	**Value:** `'System.Storage'`

24 Module System.SystemGraphOperations

Rountines for constructing meta-data graphs describing the link structure local to a give path in the Data Enviroment.

The primary use of the graphs constructed by this module are in the Graphical Browser .

24.1 Functions

MakeLocalLinkList(*Path*,
depends_on=('../System/StoredLinks/StoredImpliedLinks','../System/St...,
creates=('../System/MetaData',))

```
Given, Path, a path string describing a location in the Data Environment,
get the 2-neighborhood graph of the linklist local to that path.

It uses the linklist loaded from the Live Modules.

This function outputs the result, as a numpy sub- record array of
the Linklist, to a pickled file whose path is given by:

        MetaPathDir(Path) + 'LocalLinkList.pickle'

This path is in in the meta-data directory associated with Path.

It also outputs the file MetaPathDir(Path) + 'LiveModuleList.pickle',
which is used to cache this computation for future recomputations of it.

RETURNS:
        LinkPath = path where the locallink file was created
        (i.e. MetaPathDir(Path) + 'LocalLinkList.pickle')
```

MakeLocalLinkGraph(*Path*, *Mode*='ColDir', *ShowUses*='No', *ShowImplied*='No')

```
Takes a local link list as made by MakeLocalLinkList and turns it a
graph .svg file.  (also creates an html-ized version of the local linkgraph list.

The basic strategy of this functions is:
        1) load the local link list created in a numpy pickle file
                at the path Linkpath
        2) Cluster the nodes based on the mode:
                -- if Mode = 'ColDir' then collapse nodes within directories
                -- if Mode = 'ColPro' the collapse nodes based on their being part of
                protocols
                -- if Mode = 'All' then do no collapsing
        3) Create an html reprsentation of the link graph (with the collapsed
                clusters notated in it)
        4) Create a graph with the collapsed nodes as an object of the form

                G = [Nodes,Edges, NodeAttributes,EdgeAttributes]

        5) convert the G object into .dot and then .svg files.


ARGUMENTS:
--Path : path to make graph of local link list for
--Mode : string, which "graph quotienting" mode to use
--ShowUses : boolean, whether to include uses links or not
--ShowImplied : boolean, whether to include implied links or not

RETURNS:
[message,metapath,G,metapathhtml], where:

        message is an error message, if any (blank '' char if no error message)
        metapath = path where .svg file has been created

        G = [N,E,NAttrs,EAttrs]

        where N and E is a list of the graph nodes and edges,
        as a python list, and NAttrs and EAttrs are corresponding
        lists of node and edge attributes (for .dot graph format)

        metapathhtml = path where an html representation of the link
        list has been stored
```

MakeLinkListHtml(*LinkList*, *ClusterTags*, *inpath*, *outpath*, *Mode*, *ShowUses*, *ShowImplied*)

```
takes a data dependency linklist and a list of cluster tags, and
outputs an html representation of the linklist file (basically as
a clickable html table)

ARGUMENTS:
--LinkList = the linklist to reresent, a numpy record array
--ClusterTags = dictionary of clusters (output for example of
        the GetClusterTagDict function)
--inpath = path that the LinkList is a representation of
--outpath = place to store the resulting Html file
--Mode = Name of the cluster collapse mode
--ShowUses, ShowImplied are as in MakeLocalLinkGraph

returns:
        nothing
```

LabeledGraphFromLinks(*LinkList*, *Path*, *ClusterTags*=None, *Mode*='ColDir', *ShowUses*='No', *ShowImplied*='No')

```
Produces a graphical representation for use in producing a
.dot file representation of a Linklist>

ARGUMENTS: as is MakeLocalLinkGraph

Returns:
        object G = [Nodes,Edges,NodeAttrs,EdgeAttrs]
where
        Nodes is a python list of node names
        Edges is a python list of pairs of node names, representing node edges
        NodeAttrs is a dictionary of node attributes where:
                -- the keys are node names
                -- the value on a key 'n' is a dictionary of .dot format
                key-value attribute pairs for node 'n', e.g.
                {'color':'green','shape':'box', .... }
        EdgeAttrs is a dictionary of edge  attributes, where:
                -- the keys are edge pairs
                -- the value on a key 'e' is a dictionary of .dot format
                key-values attribute pairs for edge 'e', e.g.
                {'color':'green','shape':'box', .... }
```

EdgePropertiesSelector(*e*, *EdgeInfo*)

Technical dependency used in LabeledGraphFromLinks

EdgeTypeDeterminer(*e*, *EdgeInfo*)

Technical dependency used in LabeledGraphFromLinks

NodePropertiesSelector(*n*, *Path*, *ClusterDict*, *NodeInfo*, *Mode*, *ShowUses*, *ShowImplied*)

Technical dependency used in LabeledGraphFromLinks

NodeTypeDeterminer(*n*, *NodeInfo*, *Mode*, *ShowUses*, *ShowImplied*)

Technical dependency used in LabeledGraphFromLinks

DeleteLinkGraphs()

DeleteLocalLinkLists()

inverse(*S*)

inverts permutation described a numpy array

GetEQ(*X*, *Y*, *taglist*)

ProcessTwoDicts(*A*, *B*)

Technical dependency used in GetClusterTagDict

MaximalCP(*P*)

Technical dependency used in LabeledGraphFromLinks

SPathAlong(*p1*, *p2*)

Technical dependency used in GetClusterTagDict

GetClusterTagDict(*Path*, *LinkList*, *Mode*)

```
Given a  LinkList and a mode for collapsing the Linklist,
produce a dictionary of node cluster tags.

ARGUMENTS:
        Path -- path that the LinkList corresponds to
        LinkList -- list of Links as a numy record array
        Mode -- Mode by which to cluster the link nodes

RETURNS:
        ClusterDict, a dictionary in which:
        -- the keys are (some of the) link sources or targets in the
                LinkList (e.g. potential nodes in the graph of the linklist)
        -- value on a key is a "cluster name " which represents a set
        of nodes that will collapsed together

The "mode" determines which collapsing algorithm should be used.
```

MetaPathDir(*Path*)

WriteOutGraphDot(*G*, *outpath*)

```
Writes out graph in the [N,E,Nattrs,Eattrs] format to a .dot file

ARGUMENTS:

object G = [Nodes,Edges,NodeAttrs,EdgeAttrs], where
        Nodes is a python list of node names
        Edges is a python list of pairs of node names, representing node edges
        NodeAttrs is a dictionary of node attributes where:
                -- the keys are node names
                -- the value on a key 'n' is a dictionary of .dot format
                key-value attribute pairs for node 'n', e.g.
                {'color':'green','shape':'box', .... }
        EdgeAttrs is a dictionary of edge  attributes, where:
                -- the keys are edge pairs
                -- the value on a key 'e' is a dictionary of .dot format
                key-values attribute pairs for edge 'e', e.g.
                {'color':'green','shape':'box', .... }


outpath -- path where .dot will be put

RETURNS:
        nothing
```

LabelFunc(*n*, *Path*, *Other*, *IsCluster*)

Technical dependency of LabeledGraphFromLinks

IsProtocolPath(*s*)

Technical dependency of LabelFunc

MostCommonValue(*D*)

Given numpy array D, determines most common value D

OutsideFile(*path*)

Determines for purposes of graph coloring whether a file is an "outside" file – right now, the only kind of "outside" file that is recognized are web files. Perhaps direct connections to servers and databases should be added here for various formats eventually.

24.2 Variables

Name	Description
GraphIsTooLarge	**Value:** `500`

25 Module System.Update

```
Functions for automatic updating of data in the Data Environment by calling
scripts indicated by data dependency links.

The basic flow of information is that the functions in this module are
fed by data produced by the functions from the module
System.LinkManagement module.  Specifically, LinkManagement functions
(like GetLinksBelow) generate sequences of data dependency links
representing progations up or downward in the directed acyclic graph
formed by the set of all Data Dependency links.  These functions
can be passed parameters that have them qualify the links they return
by timestamps to show only those links that actually _need_ updating,
or be unqualified and actually show the full down- or upstream propagations.

The data dependency link sequences come in the following form:

        [L1, L2, L3 ... , LN]

where each LI is (essentially) a numpy record array of dependency
links meant to be called at stage i of an update -- because things at
stage i-1 are input dependencies to things at stage i, which are in turn
inputs to things at stage i + 1.   Within the LI, the records links
represented (essentially) as:

        (Source,Target,Script)

where "Script" is the procedure that needs to be called to produce
Target from Source.

Once the sequences of links are returned to functions here, the
UpdateLInks function calls the scripts indicated by these links,
in the rounds indicated by the sequence.

There are two basic approaches to automatic updating that are
supported by the routines provided here.

1) "Downstream Updating," in which the user asks the system to detect
changes have been made, either to data or scripts, since an update was
last run, and then propagate the results of these changes downstream
through the data dependency graph.  This functionality is implemented
by the "FullUpdate" function.

2) "Upstream Updating," in which the user specifies a target or set of
targets that he wishes to remake; then the system determines which
upstream elements have been modified since the last creation of the
target, and runs those that need updating in the proper order.  This
functionality -- which is similar in spirit to the traditional "makefile"
commands available for many code-compilation schemes -- is
implemented by the "MakeUpdated" function.
```

```
The "FindOutWhatWillUpdate" function displays what scripts will be
called, given various options to the updater, without actually calling
any updates.

The user can blend the two updating styles along a seemless
continuum.  The is done is via Automatic Updates filtering.  The
user specifies in a configuration file (whose path is set at the
AutomaticUpdatesPath environment variable) those scripts or
script-name patterns that should or should not be updated upon
downstream propation.  Then, when calling a MakeUpdated command
to a set of targets, Upstream changes are included, as well as
downstream propagations from those upstream changes that are
allowed by the Automatic Updates settings.

The advantage of Downstream Updating is that it does not require
the user to think about the implications of the changes he's making:
he merely makes changes to data and scripts, and then calls FullUpdate,
regardless of what the changes were; he does not have to remember
which files the changes are bound to affect and then make sure
those are updated.  On the other hand, this style of updating means
that the user ends up having to modify the "Automatic Updates"
settings occasionally to prevent unwanted updates from occuring,
as well to enable wanted updates to actual happen.  It also means
that he often will want to call "FindOutWhatWillUpdate" before
"FullUpdate" to make see ahead of time to see which unwated
automatic updates (if any) he may have to disable.    Downstream
Update "purists" will typically want to set the "Automatic Updates"
file to enable updating of most of what they're working on at any
given period, and then only call the two commands "FindOutWhatWillUpdate"
and "FullUpdate" repeatedly.

The Upstream style has the advantage of being more targeted -- and
requiring no interaction (if not wanted) with the automatic update
settings.   The Upstream Update "purists" -- those for example who
may be familiar with the idea of Makefiles and want to use that style
in the data analysis work -- will typically want to set the Automatic
Updates setting to contain little or nothing, and then call
MakeUpdated(Target) for whatever end-point target they ultimately
want to keep updated.

In either case, the user can _force_ updating by setting the Force options
to True (this is like the "make clean" option in make systems); or have
the system update just those things that the system thinks really need
updating.
```

25.1 Functions

MakeUpdated(*Targets*, *depends_on*=`('../',)`, *creates*=`('../Data/',)`, *Simple*=`True`, *Forced*=`False`, *Pruning*=`True`, *ProtectComputed*=`False`, *EmailWhenDone*=`None`)

```
Implements the upstream updating style

ARGUMENTS:
--Targets = List of targets to update upstream from
--Simple = if true, look for changes just at the top level of declared
        dependencies (e.g. modifitions to the directories named in the
        dependency lists);  otherwise, look for changes below (in the
        file-system sense of "below") these top-level dependency path names
--Forced = rebuild the entire upstream tree, regardless of what
        appears to be need rebuilding to handle changes.
--ProtectComputed = rebuilding downstream things that appear to
        have changed since their last official build and therefore (might be)
        corrupted.
--EmailWhenDone = email address to send report of system update to,
        upon completion (including error reports of scripts that fail)
```

FullUpdate(*Seed*=`['../']`, *Exceptions*=`None`, *Simple*=`True`, *Forced*=`False`, *Pruning*=`True`, *ProtectComputed*=`False`, *EmailWhenDone*=`None`)

```
Implements the downstream updating style.

ARGUMENTS:
--Seed = set of initial targets to go downstream from.
--Simple = if true, look for changes just at the top level of declared
        dependencies (e.g. modifitions to the directories named in the
        dependency lists);  otherwise, look for changes below (in the
        file-system sense of "below") these top-level dependency path names
--Forced = rebuild the entire upstream tree, regardless of what appears
        to be need rebuilding to handle changes.
--ProtectComputed = rebuilding downstream things that appear to
        have changed since their last official build and therefore
        might be corrupted.
--EmailWhenDone = email address to send report of system update to,
        upon completion (including error reports of scripts that fail)
```

FindOutWhatWillUpdate(*Seed*=['../'], *Exceptions*=None, *Simple*=True, *Forced*=False, *Pruning*=True, *ProtectComputed*=False)

```
Determine and print out readable report indicating which files
downstream of a seed will update (without making the actual update).

ARGUMENTS:
--Seed = set of initial targets to go downstream from.
--Simple = if true, look for changes just at the top level
        of declared dependencies (e.g. modifitions to the directories named
        in the dependency lists);  otherwise, look for changes below (in
        the file-system sense of "below") these top-level dependency path names
--Forced = act as if rebuilding the entire upstream tree, regardless of what
        appears to be need rebuilding to handle changes.
--ProtectComputed = include downstream things that appear to have
        changed since their last official build and therefore might be corrupted.
```

LinkUpdate(*Seed*, *Exceptions*=None, *depends_on*=('../',), *creates*=('../Data/',), *Simple*=False, *Forced*=False, *Pruning*=True, *ProtectComputed*=False, *EmailWhenDone*=None)

```
Given a seed and some options, call GetLinksBelow routine from LinkManagement,
and feed the result of that to the automatic updater.

This function is not usually called directly, but is called with arguments
set by either MakeUpdated or FullUpdate.

ARGUMENTS:
--Seed = set of initial targets to go downstream from.
--Simple = if true, look for changes just at the top level of declared
        dependencies (e.g. modifitions to the directories named in the
        dependency lists);  otherwise, look for changes below (in the file-system
        sense of "below") these top-level dependency path names
--Forced = act as if rebuilding the entire upstream tree, regardless of what appears
        to be need rebuilding to handle changes.
--ProtectComputed = include downstream things that appear to have changed
since their last official build and therefore (might be) corrupted.
```

UpdateLinks(*ActivatedLinkListSequence*, *Seed*, *Exceptions*=None, *Simple*=None, *Pruning*=None, *Forced*=None, *ProtectComputed*=False, *depends_on*=('../',), *creates*=('../',), *EmailWhenDone*=None)

```
This function is the main driver of automatic updating in the system.  It takes in a
sequence of sets of links, and applies the scripts indicated by those links in the
proper order.  If possible, it saves previous outputs of these scripts in a safe
place so that if a script fails, the previous output can be reverted to.  It
keeps track of success or failure along the way, and if failure occurs,
downstream updates are cancelled.   It also has a built-in diff-checker,
so that if outputs of a given script are not different from previous runs,
downstream files are merely touched (i.e., have their timestamps updated)
but not recomputed.

This function is not usually called directly by the user, but instead with its
arguments et by LinkUpdate.

Arguments:
--ActivatedLinkListSequence -- a sequence of recarrays of
        links (as produced, e.g. by GetLinksBelow).
--Seed:  the original files updates to which set off the linklist activation.
        This is used for recomputing when failure or no difference occurs.
```

printscriptrounds(*ScriptsToCall*)

25.2 Variables

Name	Description
nan	**Value:** `numpy.nan`

26 Module System.Utils

Miscellaneous utilities.

26.1 Functions

TypeInfer(*column*)

Take a list of strings, and attempt to infer a datatype that fits them all. If the strings are all integers, returns a list of corresponding Python integers. If the strings are all floats, returns a list of corresponding Python floats. Otherwise, returns the original list of strings.

Typically used to determine the datatype of a column read from a tab- or comma-separated text file.

PermInverse(*s*)

Fast invert a numpy permutation.

RecursiveFileList(*ToList*, *Avoid*=None)

```
Given list of top-level directories, recursively gets a list of files in the directories.

ARGUMENTS:
--ToList = list of top-level diretories as python list of path strings
--Avoid = List of regexps of directory name patterns to NOT look in
        e..g if a directory name matches at any level, the function will not
        look further into that directory.
```

Max(*x*)

enumeratefrom(*i*, *A*)

uniqify(*seq*, *idfun*=None)

```
Relatively fast pure python uniqification function that preservs ordering
ARGUMENTS:
        seq = sequence object to uniqify
        idfun = optional collapse function to identify items as the same
RETURNS:
        python list with first occurence of each item in seq, in order
```

FastArrayUniqify(*X*)

```
Very fast uniqify routine for numpy array
ARGUMENT:
        X = a numpy array
RETURNS:
        [D,s] where s is a permutation that will sort X, and D is the list of "first
        differences" in the sorted verion of X
        This can be used to produce a uniqified version of X by simply taking:
                X[s][D]
        or
                X[s[D.nonzero()[0]]]
        But sometimes the information of D and s is useful.
```

FastRecarrayUniqify(*X*)

```
Record array version of FastArrayUniqify.
ARGUMENT:
        X = numpy record array
RETURNS:
        [D,s] where s is a permutation that will sort all the columsn of X in some order,
        and D is a list of "first differences" in the sorted version of X
        This can be used to produce a uniqified version of X by simply taking:
                X[s][D]
        or
                X[s[D.nonzero()[0]]]
        But sometimes the information of D and s is useful.
```

DirName(*path*)

utility that gets dir name; sometimes this is the right thing to use intead of os.path.dirname itself

open_for_read(*ToRead*)

open_for_read_universal(*ToRead*)

open_for_write(*ToWrite*)

open_for_append(*ToAddTo*)

compile_expr(*expr*)

Return a code object for the given expression. Shorthand for compile(expr, 'this_filename', 'eval').

chkExists(*path*)

If the given file or directory does not exist, raise an exception

PathCompress(*path*)

redirect(*x*, *To*)

```
utility that 'redirects' a path name from to a different directory.
ARGUMENTS:
        x = path to redirect
        To = location to redirect x to
RETURNS:
        path created by taking file component of x and appending to To
```

ListArrayTranspose(*L*)

```
Tranposes the simple array presentation of a list of lists (of equal length).
Argument:
        L = [row1, row2, ...., rowN]
        where the rowi are python lists of equal length.
Returns:
        LT, a list of python lists such that LT[j][i] = L[i][j].
```

GetFunctionsDefinedInModule(*Module*)

Given live module, use inspect module to capture list of functions in module whose definition attribute is equal to the module name.

GetFunctionsMentionedInModule(*Module*)

Given live module, use inspect module to capture list of functions that appear somewhere live in the module (e.g. are loaded when the module is imported)

RedirectList(*ToRedirect*, *To*)

FixedPath(*path*)

Does some path compression, like os.path.normpath but proper for the Data Environment (consider replacing with references to os.path.normpath)

GetDataEnvironmentDirectory()

PathAlong(*a*, *b*)

returns true when path a is inside the filetree under path b.

PathStrictlyAlong(*a*, *b*)

returns true when path a is strictly insider the filetree under path b.

funcname()

returns name of function in call stack in which funcname() is being called

caller()

returns name of function in call stack which is calling the function that is calling caller()

callermodule()

returns name of module from which the function that is calling caller() was imported

TimeStamp(*T*=`None`)

deprecated TimeStamp function (should be replaced by time module formatters)

Union(*ListOfSets*)

takes python list of python sets [S1,S2, ..., SN] and returns their union

ListUnion(*ListOfLists*)

takes python list of python lists

[[l11,l12, ...], [l21,l22, ...], ... , [ln1, ln2, ...]]

and returns the aggregated list

[l11,l12, ..., l21, l22 , ...]

GetDefaultVal(*func*, *varname*, *NoVal*=`None`)

given a live python function object "func", return the default value for variable with name "varname" if it exists as a keyword variable, else return NoVal

MakeDir(*DirName*, *creates*=`()`)

is a "strong" directory maker – if DirName already exists, this deletes it first

MakeDirs(*DirName*, *creates*=`()`)

"strong" version of os.makedirs

strongcopy(*tocopy*, *destination*, *use2*=`False`)

"strong" version of copy – if destination already exists, it removes it first befire copying

delete(*ToDelete*)

unified "strong" version of delete that uses os.remove for a file and shutil.rmtree for a directory tree

Log(*s*)

Log a debug message

TemplateInstance(*templatepath*, *outpath*, ***kws*)

Apply python template at "templatepath" with substitutions from keyword arguments **kws passed in, and write out result to outpath Useful for html templating

MakeT(*r*)

If input 'r' is a comma-delimited string, return tuple split on commas, else return tuple(r)

getKalong(*LL1*, *LL2*, *k*)

Fast version of "K-along" paths.

ARGUMENTS: –LL1 = numpy array of paths –LL2 = sorted numpy array of paths –k = nonnegative integer

RETURNS: [A,B] where A and B are numpy arrays of indices in LL1 such that LL2[A[i]:B[i]] contains precisely those paths in B that are k directory levels down from LL1[i] – as path strings (no actual directory testing is done). A[i] = B[i] = 0 if no paths in LL2 are k directory levels down from LL1[i]

E.g. if

LL1 = numpy.array(['../Data/Dan_Data/', '../Users/DanYamins/','../Users/SijiaWang/'])

and

LL2 = numpy.array(['../Data/Dan_Data/NPR_Puzzle_Solutions', '../Data/Dan_Data/RandomData','../Users/DanYamins/Finance/'])

then

getKalong(LL1,LL2,1) = [A,B] = [[0,2,0],[2,3,0]]

maximalpathalong(*YY*, *ZZ*)

Fast function for determining indices of elements of YY such that they are "path along" some element of ZZ, for numpy arrays YY and ZZ. When YY[i] is path along several element of ZZ, returns index of the first occurence of the closest path. If YY[i] is not path-along any elements of Z, returns ".

getpathalong(*YY*, *ZZ*)

```
Fast version of path long for numpy arrays.

ARGUMENTS:
        LL1 = numpy array of paths
        LL2 = sorted numpy array of paths

RETURNS:
[A,B] where A and B are numpy arrays of indices in LL1 such
that LL2[A[i]:B[i]] contains precisely those paths in B that are
in the directory tree of paths in LL1[i] ( as path strings -- no actual
filesystem existence testing is done).  A[i] = B[i] = 0 if no paths in
LL2 are in the directory tree under LL1[i]

E.g. if

LL1 = numpy.array(['../Data/Dan_Data/', '../Users/DanYamins/','../Users/SijiaWang/'])

and

LL2 = numpy.array(['../Data/Dan_Data/NPR_Puzzle_Solutions',     '../Data/Dan_Data/NPR_Puzzle_Solutio
'../Data/Dan_Data/RandomData','../Users/DanYamins/Finance/'])

then

getpathalong(LL1,LL2) = [A,B] = [[0,3,0],[3,4,0]]
```

getpathalongs(*Y*, *Z*)

Returns numpy array of indices i in numpy array Y such that Y[i] is path-along some path string in Z

getpathstrictlyalong(*YY*, *ZZ*)

Version of getpathalong that requires "strictly path along"

fastequalspairs(*Y*, *Z*)

```
ARGUMENTS:
        LL1 = numpy array of paths
        LL2 = sorted numpy array of paths

RETURNS:

[A,B] where A and B are numpy arrays of indices in LL1 such that:

        LL2[A[i]:B[i]] = LL1[i].

A[i] = B[i] = 0 if LL1[i] not in LL2
```

ModContents(*obj*, *Cond*=None)

```
Modified version of Contents function, avoiding recursive inspection
of objects that satisfy condition Cond

ARGUMENTS:
--obj = BeautifulSoup object
--Cond = two-place boolean function with arugments (o1,o2) where
        o1,o2 are meant to be BeautifulSoup objects

If Cond = None this is the same as Contents
```

Contents(*obj*)

Convenience function for working with BeautifulSoup contents objects (move this somewhere else?)

ARGUMENT: –obj = BeautifulSoup object.

Given BeautifulSoup object 'obj', extract the "string contents" recursively.

fastisin(*Y*, *Z*)

fast routine for determining indices of elements in numpy array Y that appear in numpy array Z

returns boolean array of those indices

FastRecarrayEquals(*Y*, *Z*)

fast routine for determining whether numpy record array Y equals record array Z

FastRecarrayEqualsPairs(*Y*, *Z*)

IsDotPath(*s*, *path*=`None`)

Determine whether s is possible valid dot path of a python module, and is more accurante when the putative real (relative) file path is given in path. (If path argument is given this requires path to be a DataEnvironment-relative path, starting with ../)

FastRecarrayIsIn(*Y*, *Z*)

Fast routine for determining which records in numpy record array Y appear in record array Z

FastRecarrayDifference(*X*, *Y*)

fast routine for determining which records in numpy array X do not appear in numpy array Y

fastarraymax(*X*, *Y*)

```
fast way to achieve:

ARGUMENTS:
        X,Y numpy arrays of equal length
RETURNS:
        Z where Z[i] = max(X[i],Y[i])
```

fastarraymin(*X*, *Y*)

```
fast way to achieve:

ARGUMENTS:
        X,Y numpy arrays of equal length
RETURNS:
        Z where Z[i] = min(X[i],Y[i])
```

SimpleStack(*seq*, *UNIQIFY*=`False`)

Vertically stack sequences numpy record arrays. Avoids some of the problems of numpy.v_stack

SimpleStack1(*seq*, *UNIQIFY*=`False`)

Vertically stack sequences numpy record arrays. Avoids some of the problems of numpy.v_stack but is slower

if UNIQIFY set to true, only retains unique records

SimpleColumnStack(*seq*)

Stack columns in sequences of numpy record arrays. Avoids some of the problems of numpy.c_stack but is slower

RemoveColumns(*recarray*, *ToRemove*)

Given numpy recarray and list of column names ToRemove, return recarray with columns whose names are not in ToRemove

MaximalCommonPath(*PathList*)

Given list of paths, return common prefix. Like os.path.commonprefixbut proper for Data Environment purposes.

Backslash(*Dir*, *Verbose*=`False`)

Adds '/' to end of a path (meant to make formatting of directory Paths consistently have the slash)

MakeDirWithDummy(*Dir*)

makes a directory with a empty file 'dummy' in it

MakeDirWithInit(*Dir*)

makes a directory with an empty file '__init__.py' in it

GetTimeStampedArchiveName(*toarchive*)

given path string, return corresponding name that it would have in the Archive, with timestamp attached

copy_to_archive(*toarchive*, *depends_on*=`('../',)`, *creates*=`('../Archive/',)`)

copy file or directory to archive with proper archive name

move_to_archive(*toarchive*, *depends_on*=`('../',)`, *creates*=`('../Archive/',)`)

move file or directory to archive with proper archive name

CompilerChecked(*ToCheck*)

cleans a list of strings representing python regular expressions ToCheck, returning only those that are not empty and properly compile

CheckInOutFormulae(*ExpList*, *S*)

Given a list ExpList of Regular expression strings and "NOT '-prefixed regular expression strings, return list of all strings in list S that: – match at least one of the expressions in ExpList that are _not_ prefixed by 'NOT ' – match none of the expressions in ExpList that _are_ prefixed by 'NOT '

AddInitsAbove(*opfile*)

Given a python to a pytho module opfile, add an empty __init__.py file to the directory containing opfile, if not such file exists. Reset mod time of directory so it appears as if nothing as changed.

The intent of this is to allow python modules to be placed in directories in the Data Environment and then be accessed by package imports, without the user having to remember to put the '__init__.py' in the directory that the module is in. The timestamp of the containing directory is reset if the __init__.py is added to make sure that no stupid re-computations are done that make it appear as if things have changed when the havent.

DictInvert(*D*)

```
ARGUMENT:
        dictionary D
OUTPUT:
--dictionary whose keys are unique elements of values of D, and
whose values on key 'K' are lists of keys 'k' in D such that D[k] = K
```

PathExists(*ToCheck*)

```
convenient name for os function

The reason it's done this way as opposed to merely setting
        PathExists = os.path.exists
in this module is that this will disturb the system i/o intercept because this module needs to be
```

Rename(*src*, *dest*)

```
convenient name for os function

The reason it's done this way as opposed to merely setting
        PathExists = os.path.exists
in this module is that this will disturb the system i/o intercept because
this module needs to be execfiled FIRST before system_io_override.
```

IsDir(*ToCheck*)

```
convenient name for os function

The reason it's done this way as opposed to merely setting
        PathExists = os.path.exists
in this module is that this will disturb the system i/o intercept because this
module needs to be execfiled FIRST before system_io_override.
```

IsFile(*ToCheck*)

```
convenient name for os function

The reason it's done this way as opposed to merely setting
        PathExists = os.path.exists
in this module is that this will disturb the system i/o intercept
because this module needs to be execfiled FIRST before system_io_override.
```

FindAtime(*ToAssay*)

```
convenient name for os function

The reason it's done this way as opposed to merely setting
        PathExists = os.path.exists
in this module is that this will disturb the system i/o intercept
because this module needs to be execfiled FIRST before system_io_override.
```

listdir(*ToList*)

```
convenient name for os function

The reason it's done this way as opposed to merely setting
        PathExists = os.path.exists
in this module is that this will disturb the system i/o intercept
because this module needs to be execfiled FIRST before system_io_override.
```

ListAnd(*iterable*)

Return True if bool(x) is True for all values x in the iterable.

Return Value
bool

ListOr(*iterable*)

Return True if bool(x) is True for any x in the iterable.

Return Value
bool

26.2 Class BadCheckError

Error class used for raising I/O errors (maybe should be moved to system_io_override?)

26.2.1 Methods

__init__(*self*, *iofunc*, *readfiles*, *writefiles*, *Dependencies*, *Creates*)

26.3 Class multicaster

```
Class creating object that multicasts a string output stream to have both
its original desired effect and also to print any output to a log file.

typical Usage:
        sys.stdout = multicaster(sys.__stdout__,'LogFile.txt')

Then, whenever a 'print ' statement is made, output is directed both to
original stdout as well as to the logfile "LogFile.txt"
```

26.3.1 Methods

__init__(*self*, *filename*, *OldObject*, *New*=`False`)

```
ARGUMENTS:
        filename = name of file to write to
        OldObject = original output stream to multicast
        NEW = boolean which overwrites log file if true; otherwise,
        output of stream is _appended_ to 'filename'
```

__getattr__(*self*, *name*)

This is intended to answer that whenever the stdout is asked to do something other than write the function is undisturbed. If the stdout object were its own class (instead of it being merely a file \object that is being used for the purpose of output), this would be unneccesary, we'd simple subclass the stdout object.

write(*self*, *s*)

27 Package System.Web

27.1 Modules

- **DotData2Html** *(Section 28, p. 80)*
- **HTMLCreators**: Convenience functions for html templates *(Section 29, p. 81)*
- **Tabular2Html** *(Section 30, p. 82)*
- **Tabular2HtmlOld** *(Section 31, p. 83)*
- **WebRepresentations**: Unified web representation caller. *(Section 32, p. 84)*
- **py2html**: create html representation of python module *(Section 33, p. 85)*
- **text2html** *(Section 34, p. 86)*

28 Module System.Web.DotData2Html

28.1 Functions

DotData2Html(*dotDataName*, *htmlFile*)
Creates an html representation of a DotData

29 Module System.Web.HTMLCreators

Convenience functions for html templates

29.1 Functions

MakeHTMLWithFrameChooser(*PageTitle*, *ChooseBetweenFiles*, *ChooserLabelList*, *outpath*)

MakeZgrviewerAppletFile(*title*, *svgpath*, *outpath*, *target*=`'_blank'`, *depends_on*=`('../Operations/Web/zgrviewerAppletTemplate.pyt',)`)

MakeHTMLwithFrames(*inpaths*, *outpath*, *pagetitle*, *segmentnames*=`None`, *percentages*=`None`)

30 Module System.Web.Tabular2Html

30.1 Functions

Tabular2Html(*htmlFile*, *ddata*=`None`, *FileInName*=`None`, *FirstLineInSVIsHeader*=`True`, *Title*=`None`)

Creates an html representation of a Tabular data, including .data, .csv, and .tsv formats

FixCSSName(*k*)

CSSColoring(*names*, *coloring*)

ColorScheme(*NTree*, *B*, *Total*)

HeaderNotations(*names*, *sdict*)

GroupByLevel(*NTree*)

MakeLine(*names*, *L*, *sdict*)

WriteOutCSS(*ColorStyles*, *outpath*)

Tabular2HtmlString(*FileInName*, *FirstLineInSVIsHeader*=`True`)

Creates an html representation of a Tabular data, including .data, .csv, and .tsv formats

30.2 Class NameTree

30.2.1 Methods

__**init**__(*self*, *names*, *sdict*)

31 Module System.Web.Tabular2HtmlOld

31.1 Functions

Tabular2Html(*htmlFile*, *ddata*=`None`, *FileInName*=`None`, *FirstLineInSVIsHeader*=`True`, *Title*=`None`)

Creates an html representation of a Tabular data, including .data, .csv, and .tsv formats

Tabular2HtmlString(*FileInName*, *FirstLineInSVIsHeader*=`True`)

Creates an html representation of a Tabular data, including .data, .csv, and .tsv formats

32 Module System.Web.WebRepresentations

Unified web representation caller. Basically this strings together a number of different X2html functions for X = various file formats.

32.1 Functions

MakeHTMLRepresentationFast(*inpath*, *outpathdir*, *OpenFile*=`'No'`, *depends_on*=`('../System/Web/BasicHTMLTemplate',)`)

This looks at the path extension of a file and determines which (if any) web representation to apply to it.

33 Module System.Web.py2html

create html representation of python module

33.1 Functions

b(*text*)

i(*text*)

color(*rgb*, *text*)

link(*url*, *anchor*)

hilite(*text*, *bg*=`"ffff00"`)

MakePyRepresentation(*inpath*, *outpath*)

modulelink(*module*, *baseurl*=`''`)

Hyperlink to a module, either locally or on python.org

importer(*m*)

Turn text such as 'utils, math, re' into a string of HTML links.

find1(*regex*, *str*)

convert_files(*filenames*, *outpathnames*, *local_filenames*=`None`, *tblfile*=`'readme.htm'`, *openwithline*=`None`)

num_cmp(*x*, *y*)

comment(*text*)

33.2 Variables

Name	Description
id	**Value:** `r'[a-zA-Z_][a-zA-Z_0-9]*'`
replacements	**Value:** `[(r'&', '&'), (r'<', '<'), (r'>', '>'), (r'(?ms...`

34 Module System.Web.text2html

34.1 Functions

GetNumLines(*inpath*)

MakeTextRepresentation(*inpath*, *outpath*)

Main text2html function. inpath = txt file to represent outpath = html file out

if the size of the file is less than 50 MB, it's put into one HTML page if the file is > 50 MB its divided into multiple linked pages (using the MultiPage function)

MultiPage(*inpath*, *outpath*)

make multiple linked pages representing a large text file

translate(*text*, *pre*=0)

escape(*s*)

escapeq(*s*)

emphasize(*line*)

text2html(*body*)

35 Package System.config

35.1 Modules

- **SetupFunctions**: Contains functions I use for configuring the Data Environment. *(Section 36, p. 88)*

36 Module System.config.SetupFunctions

Contains functions I use for configuring the Data Environment.

36.1 Functions

GetLiveModules(*LiveModuleFilters*)

Function for filtering live modules that is fast by avoiding looking through directories i know will be irrelevant.

GetSVNFiles(*SVNPaths*=`None`)

Function for filtering files to put in the SVN repository.

37 Module System.extraction

37.1 Functions

Extract(*Seed*, *ToName*)

```
Given a seed path (or list of paths), extracts out all raw data in that path,
as well as scripts necessary to produce downstream computed data from that raw data.

The extracted object is just a single "top-level directory" containing files and
directories copied from the file system; the directory substructure within the
top-level directory replicates the directory structure of the original files.

Seed = path or list of paths given as a string, or comma-separated list of path strings,
or python list of strings.  E.g.:
                        '../Data/Dan_Data/RandomData/'
       or               '../Data/Dan_Data/RandomData/,../Data/Dan_Data/NPR_Puzzle_Solutions/'
       or               ['../Data/Dan_Data/RandomData/', '../Data/Dan_Data/NPR_Puzzle_Solutions/']

ToName = Name of top-level directory where extracted files will be copied.
This function creates the path ToName if it doesn't exist, or overwrites it if it does.
```

Integrate(*FolderToIntegrate*)

Suppose one has done an extraction (e.g., applied the Extract function) of some files in one data environment, and then taken that extracted directory and put it in the Temp directory of a new, different data environment. This function is meant to integrate the extraction into the new file system so that it mirros the placement of the files in the original data environment.

The top-level extracted directory is assumed to be in the "Temp" directory of the new data environment. This top-level directory contains a replica of a subportion of the original file system from which the extraction was done (see comments to Extract function). The Integrate function merely copies files from the extraction directory to the corresponding place in the filesystem. As it does so, it asks whether to continue if it determines that the path to which it would copy already exists in the new data environment.

FolderToIntegrate = path name of top-level extraction directory. Required to be in "Temp".

Raw(*Seed*)

```
Given a seed path (or list of paths), determines a list of all raw (non-computed)
data files in that path,
as well as scripts necessary to produce downstream computed data from that
raw data.

Seed = path or list of paths given as a string, or comma-separated list of path strings,
or python list of strings.  E.g.:
                ’../Data/Dan_Data/RandomData/’
or              ’../Data/Dan_Data/RandomData/,../Data/Dan_Data/NPR_Puzzle_Solutions/’
or              [’../Data/Dan_Data/RandomData/’, ’../Data/Dan_Data/NPR_Puzzle_Solutions/’]


Returns: [A,B] where
--A = list of path names in the data environment; this list contains all raw
(non-computed) data files required to produce the computed data files in Seed
paths, as well as the paths of all scripts that need to be run to produce the
computed files.
--B = boolean which is True when all paths listed in A actually exist in the file system.
```

CopyOut(*FilesToCopy*, *ToName*)

Copies out files from one location to a ”top-level target directory” within the Temp Directory of the Data Environment, preserving the original internal directory structure.

FilesToCopy = List of files to copy, given as a python list of path names.

ToName = name of top-level directory to which the files in FilesToCopy will be copied. The top-level directory is given as a relative path, and will be placed in the Temp directory of the data environment.

If ToName does it exist, this function creates it. If it does exist, it will overwrite it.

As files are copied into the ToName directory, original internal director structure is replicated, e.g. if a file File.txt has path ../A/File.txt relative to Temp, then (if it doesn’t yet exist in ToName), an empty directory called ToName/A will be created, and the File.txt will compied to it.

CopyIn(*ToCopy*, *Target*, *overwrite*=`False`)

ToCopy = a path name. Target = a path name overwrite = Boolean

This function copies the contents of ToCopy to Target, preserving whatever file structure is in ToCopy (e.g. just the file if it is a file, and the directories in it if it is a directory.)

If Target does not exist, it creates it. If it already exists, CopyIn overwrites Target if overwrite argument is True; if overwrite argument is False, the function aborts.

38 Module System.system_io_override

```
Intercept system i/o functions and analyze stack that made i/o calls.

One of the main purposes of the Data Environment to understand
and manage data dependencies.  At a basic level, these data
dependencies involve reading and writing files to disk.  To this end,
it is useful to be able to determine exactly which files are being read
and written, realtime -- by during processes of individual users doing
exploratory data analysis as well as by automatic system updates.
One wants not only to know which i/o streams are being started, but
also which high-level processes and functions are doing the reading and writing.

This module provides routines for:
        1) intercepting i/o streams at a low level
        2) analyzing the call stack above those streams
        3) integrating the information from 1 and 2 to:
                -- store data about which files were read/written, and
                therefore enable run-time determination of _undeclared_
                data dependency links
        and
                -- impose controls on which files can be written by which
                processes, thereby enabling the enforcement of _declared_
                data dependency links.

The idea behind the imposition of controls is not to give the Data Enviroment
an account security interface whereby Data Environment users can be
flexibly prevented by Data Environment administrators from doing things
with files that don't belong to them. (Such things are probably better
administered at the operating system level by real user accounts.)
Instead, the idea is to allow the users to impose controls on themselves
so that they, and others using their data, can be assured that the
LinkLists that the Data Environment builds from the users' declarations
are an accurate reflection of the way the data was actually produced
-- and have errors raised when dependency violations occur.  This allows
users to easily see where they've forgotten to declare dependencies properly,
and trivially know how to remedy the issue.

The majority of the routines in this file are the system i/o intercepts themselves.
The idea in each case is to implement for a standard i/o function, the
following three-step "redefinition" procedure:
        1) take the standard __builtin__ I/O functions and save it to a "hold name"
                (e.g. by prefixing "old_" to the name)
        2) redefine a new i/o function with PRECISELY the same interface as
                the old one, and whichn calls the function at the "holding name" after
                first having passed the inputs and outputs through an I/O interception
                and check that uses information from the GetDependsCreates() call
                stack analyzer function.
        3) redefine the system's standard I/O name as the new i/o function.
```

```
Step 2) can either implement:
        a) a "control" -- meaning that an i/o function that is called without being
                covered by a declared dependency will raise a "BadCheckError" that
                prevents the I/O operation from occuring and prints information about the
                violation,
or
        b) a "logging" operation -- meaning that data about the i/o request, together
                with the stack inforamtion about where it was made and whether it was
                consistent with declarations -- is stored in some standard format and place.
-- and of course, it can do both.

The slightly complex 3-step redefinition procedure is used to prevent
problematic infinite recursions.  It also must be handled with care because,
once these intercepts are activated, ALL calls to these system functions
will be passed through the intercepts.   This means that:
        -- protections can't be set too widely,
        -- interfaces of the functions must be preversed EXACTLY,
so that programs indirected called by the user as dependencies to
other functions, and that require resources "outside" the system, are
not bizarrely restricted from performing their work. (This is especially
the case for automatic logging and other interactive features of enhanced
python shells like iPython, which I tend to want to use.)
```

38.1 Functions

IsntProtected(r)

If the environment variable "PROTECTION" is set to "ON" (see comments in i/o GetDependsCreates), this function looks to see whether an environment variable called "ProtectedPathsListFile' has been set. This is a path to a user configuration file which would list, in lines, the list of directories on which the user wishes i/o protection to be enforced. If no such path variable is set, the system looks uses DataEnvironmentDirectory as the default, e.g everything in the DataEnvironment would be protected if the PROTECTION variable is set at 'ON'.

GetDependsCreates()

```
This function performs call stack analysis looking to capture and
aggregate the settings of the two protected key-word variables
"depends_on' and "creates" that are present up the stack.

RETURNS:
--[dlist,clist] where:
If the "PROTECTION" variable is set to 'ON':
        dlist is the aggregate tuple of depends_on values seen
        up the stack tree, and clist is the aggregate tuple of
        creates values.

        If 'SystemMode' environment variable is set at 'Exploratory',
        it adds '../' to the dlist, (which will be interpreted to allow
        unlimited read access) -- the idea being that at the user prompt,
        mostly the user will want to be able to at least read whatever
        files he/she wants, even he/she cannot then write back to those
        paths.

else:
        clist and dlist are set to '../', which will be interpreted by system
        i/o intercepting functions that call GetDependsCreates to mean
        that unlimited read/write access is granted.

'../Temp' is appended to clist and dlist in all cases, to allow the user
to read and write whatever is desired in working Temp directory.

Probably a more flexible and less hard-coded way of allowing the
user to set these read-write protections would be useful in the future.
```

system_open(*ToOpen*, *Mode*=`'r'`, *bufsize*=`1`)

system_copy(*tocopy*, *destination*)

system_copy2(*tocopy*, *destination*)

system_copytree(*tocopy*, *destination*, *symlinks*=`False`)

system_mkdir(*DirName*, *mode*=`0777`)

system_makedirs(*DirName*, *mode*=`0777`)

system_rename(*old*, *new*)

system_remove(*ToDelete*)

system_rmtree(*ToDelete*, *ignore_errors*=`False`, *onerror*=`None`)

system_exists(*ToCheck*)

system_listdir(*ToList*)

system_isfile(*ToCheck*)

system_isdir(*ToCheck*)

system_getmtime(*ToAssay*)

system_getatime(*ToAssay*)

38.2 Variables

Name	Description
old_open	**Value:** `__builtin__.open`
old_copy	**Value:** `shutil.copy`
old_copy2	**Value:** `shutil.copy2`
old_copytree	**Value:** `shutil.copytree`
old_mkdir	**Value:** `os.mkdir`
old_makedirs	**Value:** `os.makedirs`
old_rename	**Value:** `os.rename`
old_remove	**Value:** `os.remove`
old_rmtree	**Value:** `shutil.rmtree`
old_exists	**Value:** `os.path.exists`
old_listdir	**Value:** `os.listdir`
old_isfile	**Value:** `os.path.isfile`
old_isdir	**Value:** `os.path.isdir`
old_mtime	**Value:** `os.path.getmtime`
old_atime	**Value:** `os.path.getatime`

Index