The Stingray Schema-Based File Reader

Release 4.4.2

S. Lott

May 14, 2014

CONTENTS

The Stingray Reader tackles four fundamental issues in processing a file:

- How are the bytes organized? What is the Physical Format?
- Haw are the data objects organized? What is the Logical Layout?
- What do the bytes *mean*? What is the Conceptual Content?
- How can we assure ourselves that our applications will work with this file?

The problem we have is that the schema is not always bound to a given file nor is the schema clearly bound to an application program. There are two examples of this separation between schema and content:

- We might have a spreadsheet where there aren't even column titles.
- We might have a pure data file (for example from a legacy COBOL program) which is described by a separate schema.

One goal of good software is to cope reasonably well with variability of user-supplied inputs. Providing data by spreadsheet is often the most desirable choice for users. In some cases, it's the only acceptable choice. Since spreadsheets are tweaked manually, they may not have a simple, fixed schema or logical layout.

A workbook (the container of individual sheets) can be encoded in any of a number of physical formats: XLS, CSV, XLSX, ODS to name a few. We would like our applications to be independent of these physical formats. We'd like to focus on the logical layout.

Data supplied in the form of a workbook can suffer from numerous data quality issues. We need to be assured that a file actually conforms to a required schema.

CHAPTER

ONE

CONTENTS

1.1 Introduction

Given a workbook – or any other “flat” file – how is it organized? What does it *mean*?

How can we ignore details that are merely physical? When is a change merely a logical layout variant with no semantic impact? How do we isolate ourselves from these layout variants?

There are two use cases that we want to focus on.

- **Extract Transform and Load (ETL)**. We’ve got a file of data in the form of a workbook (“spreadsheet”) file or perhaps a flat (delimiter-free) COBOL file.
- **Analysis**. Also known as Data Profiling. We’ve got a file of data and we want to examine the data for frequencies, ranges and relationships. A common use for this is to “validate” or “unit test” a file before attempting to process it.

In these cases, we don’t want our applications to depend on the physical file format. We want to work equally well with CSV or XLSX versions of a file.

We’d like something like this to transparently handle workbook files in a variety of formats.

```
def process_sheet( sheet ):
    ``’’``Separated to facilitate unit testing’’``’’
    counts= defaultdict( int )
    for rows in sheet.rows():
        process row
    return counts

def main(args):
    for input in args.file:
        with workbook.open_workbook( input ) as source:
            for name in source.sheets():
                sheet= source.sheet( name,
                    sheet.EmbeddedSchemaSheet,
                    loader_class=schema.loader.HeadingRowSchemaLoader )
                counts= process_sheet( sheet )
                pprint.pprint( counts )
```

Note that this specifies a number of things explicitly.

- `for name in source.sheets()` claims that *all* sheets have valid data. When this is not the case, more sophisticated name filtering needs to be inserted into this loop.

- sheet.EmbeddedSchemaSheet states that the sheet has the schema embedded in it. For example, the headers in the first row of the sheet. We can use sheet.ExternalSchemaSheet to specify that an external schema must be loaded. For example, a separate sheet in this workbook or perhaps a separate file.
- schema.loader.HeadingRowSchemaLoader states that the schema is the first row of the sheet. We can write other parsers for other sheets with more complex or atypical layouts.

Beyond physical format transparency, we want applications that are flexible with respect to the the logical layout. We'd like to be adaptable to changes to the names, number, order and type of columns without a significant rewrite. We'd like to isolate the column names or positions from the rest of our application processing.

We can do this by definining small "builder" functions which isolate the logical layout from the rest of the application.

We'd like something like this.

```
def build_record_dict( aRow ):
    return dict(
        name = aRow['some column'].to_str(),
        address = aRow['another column'].to_str(),
        zip = aRow['zip'].to_digit_str(5),
        phone = aRow['phone'].to_digit_str(),
    )
```

The column names and data conversions are isolated to this function only. As the source schemata evolve, we can update an application to add variations on this function.

From this, we can build Python objects using the following:

```
def build_object( record_dict ):
    return ObjectClass( **row_dict )
```

We can combine the two steps like this:

```
def object_iter( source ):
    for row in source:
        rd= self.build_record_dict( row )
        yield self.build_object( rd )
```

This design breaks processing into two parts because the logical layout mapping from workbook rows to Python objects is never trivial. This is subject to change with minimal notice. Experience indicates that it's better to break the processing into two discrete steps so that transformations are easier to manage and extend.

We can then use this iterator to process rows of a sheet.

```
def process_sheet( sheet ):
    counts= defaultdict( int )
    for app_object in object_iter( sheet.rows() ):
        process the app_object
    return counts
```

1.1.1 Deeper Issues

Processing a workbook (or other flat file) means solving two closely-related schema management problems.

- Decoding the "Physical Format". Format is the organization of bytes from which we can decode Python objects (e.g., decode a string to lines of text to atomic fields). A workbook should make the physical format irrelevant.
- Mapping to the "Logical Layout". That is, the semantic mapping from Python items (e.g., strings) to meaningful data elements in our problem domain (e.g., customer zip codes.)

Both of these are implementation details for the *Conceptual Content*. The conceptual schema is the *meaning* behind the encoded data.

Often, the physical format issue is addressed by using a well-known (or even standardized) file format: CSV, XLSX, XML, JSON, YAML, etc. We'll start with just the workbook formats CSV, XLSX and ODS.

We'll ignore JSON and YAML for the time being. We should be able to extend our model from highly structured data to semi-structured data, including JSON, YAML, and even documents created with outliner tools.

The logical layout issue is not as easy to address as physical format. Why not?

- **Bullfeathers**. Also known as Semantic Heterogeneity. You call them "customers" because they have a contract and pay money for services. Marketing, however, calls their prospects "customers" even though there is no contract in place.

  Same word. Different semantics. Yes. It's a "problem domain" issue. No, it's not solvable.

  All we can do is cope. This means having enough flexibility to handle the semantic issues.

- **Buffoonery**. Is "CSTID" the "customer id number"? Or is it the "Commercial Status ID"? Or is it the "Credit Score Time Interval Delta"? Or is it something entirely unrelated that merely happens to be shoved into that field?

  We'll need a design that has the flexibility to cope with variant abbreviations for column names.

- **Bad Management**. At some point in the past, the "Employees Here" and "Total Employees" were misused by an application. The problem was found–and fixed–but there is some "legacy" data that has incorrect data. What now?

  This leads to rather complex designs where the mapping from source to target is dependent on the source data itself. Or the context for the source data.

- **Bugs**. The field named "Effective Date" is really the *Reported Date*. The field name "Supplied Date" is really the *Effective Date*. The field labeled "Reported Date" isn't populated consistently and doesn't seem to have any meaning. Really.

  This is also a kind of mapping issue. It's analogous to the "Buffoonery" issue, above. Flexible mappings are essential.

There is an underlying *Conceptual Schema*. However, it has numerous variant implementations, each with a unique collection of errors and anomalies.

1.1.2 Directions

We need to have a representation for a schema that allows an application *some* flexibility. We have four plus one views of an acceptable solution to the schema representation problem.

- The use cases need to make sense to the actors. To retain the user's terminology, a data dictionary can help interpret a use case in spite of the user's imprescise terminology. Additional aliases for names and descriptions will be essential.
- The logical view needs to focus on the problem domain, eschewing physical format issues. (If mentioned at all, format should be little more than a stereotype attached to a class model.) The logical view must allow variability in the logical layout of the workbook or file.
- The deployment view should allow multiple applications to properly follow the Don't Repeat Yourself (DRY) principle and share a schema definition. Schemata are precious, centralized components which bridge data and processing. If a schema is bound with the data (to the extent possible), then it can be reused more effectively
- The processing view should allow a single application program to work sensibly with files that have the same logical layout but different physical formats. A single conceptual schema may have variant logical layouts (e.g., column name spelling changes). This layout variants should be tolerated by a single application.

- The physical view is focused on file processing: either workbooks or flat data files.

1.1.3 Misdirections

We have to be cautious of trying too hard to leverage the `csv` module's row-as-dictionary view of data. Above, we suggested that the `csv` representation was ideal.

The `csv.DictReader` approach – implicitly creating a dict instead of a sequence – fails us when we have to work with COBOL or Fixed Format files. These non-spreadsheet files may not be close to First Normal Form. COBOL files have repeating groups which require numeric indexes in addition to column names.

For semi-structured data (JSON, YAML our an outline) there are fewer constraints, leading to a more complex normalization step and possible row validation rules.

Further, the `csv` approach of **eagerly** building a row doesn't work for COBOL files because of the `REDEFINES` clause. We can't reliably build the various "cells" available in the COBOL schema, since some of those values may turn out to be invalid. COBOL requires lazily building a row based on which REDEFINES alias is relevant.

We also have to avoid the attractive nuisance of trying to create a "data mapping DSL". This is seductive because a data mapping is a triple of target, source and conversion.

```
target = source.conversion()
```

Since this is – effectively – Python code, there's no real reason for creating a DSL. Just use Python.

Historical Solutions

> "Those who cannot remember the past are condemned to repeat it." –George Santayana

We'll look at four solutions in their approximate historical order.

The COBOL Schema Solution is still relevant to modern Python programmers.

The DBMS Schema Solution is available, but isn't compeletly universal. It doesn't apply well to files outside the database.

The CSV Schema Solution often introduces more problems than it solves.

There is an XML Non-Solution. While XML is relevant, it is not the *trivial* solution that some folks seem to hope for. At best, it offers us XSD, which may be too sophisticated for the problem we're trying to solve.

For semi-structured data (JSON, YAML and outlines), we need more than a simple schema definition. We need processing rules to reformat and validate the inputs as well.

1.1.4 COBOL Schema Solution

A significant feature of the COBOL language is the Data Definition Entry (DDE) used in the data and environment divisions of the source. This was a hierarchical structure that defined the various items in a single record of a file.

Hierarchical. Like XML.

Best practice was essentially DRY. Developers would keep the definitions as separate modules under ordinary source code control.

Every application that worked with a given file would import the DDE for that file.

Clearly, the binding between schema and file is a two-step operation. There's a compile-time binding between schema and application. There's a run-time binding between application and file.

Just as clearly, this is subject to all kinds of mysterious problems when schema definition modules where cloned and then modified, leaving it unclear which is correct. Also, when a schema definition was modified and not all programs were properly recompiled.

Since the schema wasn't formally bound to the file, it was particularly easy to have files without any known schema. Ideally, the file name included some schema hint.

What's relevant for Python programmers is the central idea of a schema being

1. External to any specific application.
2. DRY.

To this, we would like to assure that the schema was bound to the relevant files. This is much more difficult to achieve in practice, but there are some approaches that can work well.

We'll look at this closely, below.

1.1.5 DBMS Schema Solution

A Database Management System (DBMS) – whether relational or hierarchical or columnar or networked or whatever – addresses the problems with flat files and the separation between application program, physical format, logical layout, and operating system file.

The DBMS should provide us a logical/physical schema separation

The physical files are managed by the DBMS. Our applications are now independent of physical file structure (and indepenent even of OS.)

The logical "table structure" (or whatever is offered) is distinct from the files. The logical schema it tightly bound to the data itself. When using SQL, for example, the column names and data types are available as part of the execution of each SQL query.

This binding between data is schema is ideal.

Sadly, it doesn't apply to files. Only to databases as a whole.

If file transfers are replaced with SQL queries (or web services requests) then schema is discoverable from the database (or web service). However, the problem we're addressing here is file transfer. So this solution is inappropriate except as a guidepost.

1.1.6 CSV Schema Solution

A workbook (or "spreadsheet") may or may not have schema information.

The workbook may have any of a large number of physical formats: .XLS (i.e., native) .XLSX or .CSV. The format is irrelevant to the presence or absence of a schema.

One common way to embed schema information is to assure that the first row of each sheet within a workbook has the schema information. This is the "column titles" solution.

- This isn't done consistently. Column titles are omitted. Sometimes the titles occupy multiple rows. Sometimes the sheet itself has a heading/body format where there's irrelevant rows which must be filtered out.
- This is subject to all kinds of buffoonery. Column titles can be tweaked manually.
- The data type information is bound to each cell; the column title provides no usable type information.

  Exacerbating this is the way that anything number-like becomes a floating-point number. Zip codes, punctuation-free phone numbers and social security numbers, etc., all become floating-point numbers. This means they lose their leading zeroes, making zip-code matching particularly painful.

Another common case is–of course–to omit schema information entirely. This devolves to the COBOL Schema Solution where the schema information must be incorporated as a separate, reusable module of some kind.

A less common case is to include a separate sheet in the workbook (or worse, a separate file) with schema information. A separate sheet in a workbook is at least bound with the data. A separate schema description file (even if bound in a ZIP archive) can get unbound from the data.

While there are numerous problems, workbooks are a very common way to exchange data. It's not sensible to pretend they don't exist. We can't paraphrase Jamie Zawinski and say "Some people, when confronted with a problem, think 'I know, I'll use [a spreadsheet].' Now they have two problems." We do need to process this data irrespective of the issues.

We need a suitable, Pythonic solution to the schema problem when confronted with data in a spreadsheet.

1.1.7 XML Non-Solution

Weirdly, XML fans will state the XML is "self-defining". Their claim is that somehow this *solves* the schema problem. This statement can be misleading.

For our purposes, XML is a physical format. An XML document without a referenced (or embedded) XSD lacks any semantic information. Even then, an XSD can be helpful, but not sufficient. The type definitions in an XSD could be unclear, ambiguous or simply wrong.

An XSD document can be a good, useful schema definition. It can also be riddled with buffoonery, bad management and bugs. It isn't magically perfect. It merely frees us from physical format considerations.

We can – to an extent – leverage elements of PyXSD (http://pyxsd.org) to create Python classes from an XSD schema. This package could help by providing a standard class structure for a schema defined in Python. In particular, the `pyxsd.schema` module contains some of what we're going to use.

XSD is a bit too complex for this problem domain. Spreadsheets don't make use of the sophisticated modeling available in XSD. A spreadsheet is a flat list of of a few simple types (float, string, date-encoded-as-float.) A COBOL DDE is a hierarchy of of a few simple types (display, comp-3). All we can really do is borrow concepts and terminology.

1.1.8 Summary

Physical format independence is available with some file formats. Sadly, others – which are still in active use – require a great deal of schema information merely to decode the physical format.

Logical layout is generally a feature of the application program as well as the data. Even in a SQL-based data access, the column names in a `SELECT` statement amount to a binding to a schema.

While we can make some kinds of simple applications which are completely driven by metadata, the general solution to information processing remains a proper programming language.

Therefore, we need to have application programs which can tolerate physical format changes without complaint.

We would also like an application program that can work with "minor" variations on a logical layout. That is, the order of columns, or minor spelling changes to a column name can be handled gracefully.

We'd like our batch processing applications to have a command-line interface something like this.

```
python -m some_app -l layout_2 some_file.xyz
```

The `-l layout_2` provides logical layout information. This defines the "application-level" schema information.

The `some_file.xyz` could be `some_file.xls` or `some_file.ods`, allowing transparent changes to physical format.

Schema Representation Considerations

There are several sensible way to represent schema information. Each has a problem. We're forced to chose the lesser of the various evils.

1. One (or more) rows in each workbook sheet that provides the attribute name. The type is left implicit. Other information (e.g., offset or size) is part of a physical format: it's only needed for fixed-format files. The "row-header schema" is casual and could involve numerous kinds of technical errors (missing names, duplicate names, incorrect names.)
2. A distinct workbook sheet that lists name, data type information for each attribute. Offset and size for fixed format files can also be provided. This can involve numerous kinds of technical errors (missing attributes, duplicate attribute names, incorrect attribute descriptions.)
3. A Python module that's built from source information. This allows us to trivially `import schema.foo` and have lots of cool classes and functions in the `schema.foo` module. This pushes the envelope the DRY principle because the module would have to be built from other source data.
4. Some standardized metadata format. XSD (or even XMI) pops into mind. These are detailed and potentially useful. Experience shows, however, that schema definitions are almost universally provided as workbook sheets. An external schema in a standard notation pushes the envelope on the DRY principle and also is vanishingly rare in practice.

Items 1 and 2 (workbook-based schema) cover over 99% of the cases in practice. While the data is casual and error-prone, it's readily available and consistent with DRY.

Option 3 (a Python module) – while cool – breaks the DRY principle. Refreshing a Python schema when a source `.XLS` document changes is annoying because we're tweaking the way `import` works. The imported schema module has to confirm that it's newer than the files it's derived from and possibly rebuild itself *automagically* after the schema source is touched.

Option 4 is rarely used in practice. In the rare cases when an organization will consent to providing XSD files, they're often prepared separately from the data and do not actually reflect the application software that is used to build the data file.

Plus, we should also be able to extend our schema representation to cope with COBOL DDE's. These have a place, since legacy conversion and data extraction is still sometimes required.

Finally, we may be forced to deal with data in semi-structured formats like JSON, YAML or an outline.

Our best approach is to load schema information from source every time it's needed. There are two paths:

- Either we'll parse the embedded schema buried in each sheet of a workbook,
- or we'll load an external schema definition from a file.

1.1.9 The COBOL Issues

When dealing with "Flat Files" from legacy COBOL problems, there are several additional problems that need to be solved.

1. The files have a fixed field layout, without delimiters. This means that the offset of each field must be used to decompose the record into its individual elements.
2. Numeric fields can have an implied decimal point, making it difficult to determine the value of a string of digits. The COBOL DDE is essential for parsing the file contents.
3. COBOL can make use of numeric data represented in a variety of "Computational" forms. The "Computational-3" ("COMP-3") form is particularly complex because decimal digits are packed two per byte and the final half-byte encodes sign information.

4. The string data may be encoded in EBCDIC bytes.
5. COBOL encourages the use of data aliases (or "unions") via the `REDEFINES` clause. Without the entire unviverse of COBOL programs that work with a given file, the general handling of `REDEFINES` data elements can become an insoluable problem. Only lazy field access can work; eager creation of individual cell values is doomed because a `REDEFINES` alternative may be defined over invalid data.
6. COBOL has an `OCCURS DEPENDING ON` (ODO) feature where one attribute determines the size of another attribute. This means the data of every attribute after the ODO attribute has a location which varies. The positions within the flat file cannot be computed statically.

Generally, COBOL files are defined by a "Data Definition Entry" (DDE) that provides the record layout.

Developing the offsets to each field manually is tedious and error-prone work. In the presence of ODO, the locations become expressions based on the current record's data.

It's essential to parse the DDE, which has the original COBOL source definition for the file. A schema can be built from the parsed DDE.

1.2 Licensing

See http://creativecommons.org/licenses/by-nc-sa/4.0/

This work is made available under the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International (CC by-nc-sa 4.0) license.

You are free to:

> **Share** — copy and redistribute the material in any medium or format
>
> **Adapt** — remix, transform, and build upon the material
>
> The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:

> **Attribution** — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
>
> **NonCommercial** — You may not use the material for commercial purposes.
>
> **ShareAlike** — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.
>
> **No additional restrictions** — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

If you want to use this for commercial purposes, contact the author. Commercial use will involve a proper licensing fee and a different license.

1.3 The `stingray` Package

The `stingray` package implements a Schema-based File Reader. This allows us to use flat-file and workbook data from Python without having to clone an application for each physical file format or logical layout.

It also allows us to use semi-structured data like JSON, YAML or an outline. This can be handled consistently with structured data.

This package includes

- A definition for "workbook", "sheet" and "cell". This can subsume `csv`, `xlrd` as well as XML parsers for XLSX and ODS files. This makes the physical format transparent to an application.
- A definition for a schema. Not a complex XSD, but the limited, flat schema appropriate for rows in sheets of a workbook in approximately First Normal Form. This is extended to handle the simple hierarcical COBOL features.
- Classes to load a schema that's embedded either in a sheet or in a separate file.
- A COBOL schema loader as an extension to the default loaders.

This depends on Python 3.3.

The structure of the `stingray` package is as follows.

- `__init__.py`. Some essential overhead. See *stingray __init__.py*.
- `cell.py`. *Cell Module – Data Element Containers and Conversions* defines the `Cell` class hierarchy. Imported as `stingray.cell`.
- `sheet.py`. *Sheet Module – Sheet and Row Access* defines the `Sheet` class hierarchy that supports sheets with embedded as well as external schema. Imported as `stingray.sheet`.
- `workbook.py`. The *Workbook Package – Uniform Wrappers for Workbooks* families. Imported as `stingray.workbook`.
- `schema`. This package defines a schema and schema loaders.
  - `__init__.py`. *Schema Package – Schema and Attribute Definitions*. Imported as `stingray.schema`. This is the generic, flat schema and superclasses for the more complex COBOL schema.
  - `loader.py`. *Schema Loader Module – Load Embedded or External Schema*. A loader for the generic, flat schema. Imported as `stingray.schema.loader`. Applications will often extend schema loaders to handle peculiar formats or multi-line headings or other workbook formatting.
- `cobol`. This package extends a schema and schema loaders to handle COBOL files.
  - `__init__.py`. *The COBOL Package*. Imported as `stingray.cobol`. These are extensions to `stingray.cell`, `stingray.sheet` and `stingray.workbook`.
  - `loader.py` *COBOL Loader Module – Parse COBOL Source to Load a Schema*. A loader for COBOL-syntax schema. Imported as `stingray.cobol.loader`.
  - `defs.py`. *COBOL Definitions Module – Handle COBOL DDE's*. Base definitions used by both `__init__.py` and `loader.py`.
- `snappy`. This module is a minimal implementation of a reader for files written with Snappy compression.
- `protobuf`. This module is a minimal implementation of a reader for objects represented using protobuf.

1.3.1 stingray `__init__.py`

This is approximately pure overhead required to make a Python package.

```
"""stingray --  Schema-based File Reader helps handle physical format and logical
layout of workbooks and flat files.

Requires Python 3.3, xlrd 0.9.2
"""
__version__ = "4.4.2"
```

1.4 Cell Module – Data Element Containers and Conversions

The point of a `cell.Cell` is two-fold.

- **Capture**. That is, decode the information in the source file into a Python object that represents the source spreadsheet value. For XLS or XLSX formats, there are a variety of cell data types. For CSV, all cells are text. For a fixed format file, we may have to exploit the physical format information to properly decode the bytes or characters. We may even have to cope with EBCDIC or packed decimal conversions.
- **Convert**. Provide the cell value coerced to another type.

1.4.1 Capture Use Case

The **Capture** use case is defined by our physical formats. `xlrd` identifies the follow cell types found in XLS workbooks.

Type symbol	Type number	Python value
XL_CELL_EMPTY	0	empty string ‘’
XL_CELL_TEXT	1	a Unicode string
XL_CELL_NUMBER	2	float
XL_CELL_DATE	3	float
XL_CELL_BOOLEAN	4	int; 1 means TRUE, 0 means FALSE
XL_CELL_ERROR	5	int representing internal Excel codes; for a text representation, refer to the supplied dictionary error_text_from_code
XL_CELL_BLANK	6	empty string ‘’. Note: this type will appear only when open_workbook(..., formatting_info=True) is used.

An XLSX (per ECMA 376, section 18.18.11) or ODS provides a similar list of cell types. The data is always encoded as a proper string that can be converted (if necessary) based on the type code.

Enumeration Value	Description
`b` (Boolean)	Cell containing a boolean.
`d` (Date)	Cell contains a date in the ISO 8601 format.
`e` (Error)	Cell containing an error.
`inlineStr` (Inline String)	Cell containing an (inline) rich string, i.e., one not in the shared string table. If this cell type is used, then the cell value is in the is element rather than the v element in the cell (c element).
`n` (Number)	Cell containing a number.
`s` (Shared String)	Cell containing a shared string.
`str` (String)	Cell containing a formula string.

Dates formatted as strings are – always – a problem. There’s no generic solution. An application may need to write suitable extensions to handle this. See below under Conversion Functions.

We should depend on the `locale` module to provide proper format strings for converting between date and string.

A Numbers spreadsheet appears to have the following cell types. Some of the tags appear to have no value, so their purpose is unclear.

Tag	Description
d (Date)	Cell containing a date
f (Formula)	Cell containing a formula
g (Empty)	Two or more empty cells
n (Number)	Cell containing a number
o (?)	One empty cell
s (?)	
t (Text)	Cell containing text
pm (Popup Menu)	A popup menu of otherc cell values

1.4.2 Convert Use Cases

There are several use cases for output conversion (or "transformation").

- Trivial. `float`, `str` or an empty cell. Essentially, we're using the captured data type directly.
- Easy. `float` or `str` to `decimal`. Generally, currency fields are stored as `float` in the workbook; this needs to be coverted to `decimal` to be useful.
- Obscure. `datetime` based on `float`. `xlrd` handles this elegantly.
- Variable. `datetime` based on `str`. The variability becomes rather complex. It's also application-specific, since it depends on the source of the data.
- Horrible. Digit strings. US Zip codes. Social Security Numbers. Phone numbers without punctuation. These are digit strings which a spreadsheet application may transform to a floating point number; these need to be rebuilt as proper digit strings with leading zeroes.

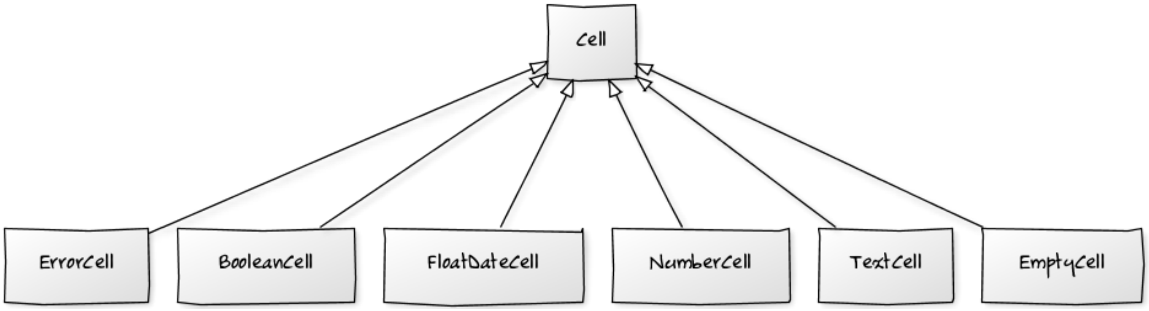
We'd like code that looks like these examples:

```
"{0} has {1}".format( foo.to_str(), foo.to_float() )

today= bar.to_datetime().date()

zip_code.to_digit_str(5)
```

This **Convert** aspect of a `cell.Cell` is part of *using* the logical layout. We'll address that under *The Stingray Developer's Guide*, below.

1.4.3 Model

```
http://yuml.me/diagram/scruffy;/class/
#cell,
[Cell]^[EmptyCell],
[Cell]^[TextCell],
[Cell]^[NumberCell],
[Cell]^[FloatDateCell],
[Cell]^[BooleanCell],
[Cell]^[ErrorCell].
```



1.4.4 Overheads

We required `xlrd` from http://www.lexicon.net/sjmachin/xlrd.htm

This is the easiest way to read legacy Microsoft `.XLS` files.

```
"""stingray.cell -- Defines Cell as the atomic data element in a sheet
of a workbook.

A cell has a value, it's part of a workbook.
"""

import locale
import decimal
import datetime
import xlrd
import time
from collections import Hashable
```

Just to be sure that any locale-based processing will actually work, we establish a default locale.

```
locale.setlocale(locale.LC_ALL, '')
```

1.4.5 Cell

class `cell.`**`Cell`**

The `cell.Cell` class hierarchy extends this base class. Note that we have a relatively short list of built-in conversions. For more complex, application-specific conversions, the raw `value` is available as a property.

```
class Cell( Hashable ):
    """A class hierarchy for each kind of Cell."""
    def __init__( self, value=None, workbook=None ):
        """Build a new Cell; the atomic data element of a  workbook.

        :param _value: the proper Python object, correctly converted from the source.
        :param workbook: the :py:class:`workbook.Workbook` that created this Cell.
            This is largely used for Excel date conversions, but there
            could be other context needs.
        """
        self._value, self.workbook = value, workbook
    def __repr__( self ):
        return "{0}({1!r})".format(
            self.__class__.__name__, self._value )
    def is_empty( self ):
        return self._value is None
    def to_int( self ): return NotImplemented
    def to_float( self ): return NotImplemented
    def to_decimal( self, digits=None ): return NotImplemented
    def to_str( self ): return NotImplemented
    def to_datetime( self, format=None ): return NotImplemented
    def to_digit_str( self, len=5 ): return NotImplemented
```

One feature of a cell that's required when we do data profiling is to create a usable hash from the cell class and raw data value.

```
def __hash__( self ):
    return hash(self._value) ^ hash(self.__class__)
def __eq__( self, other ):
    return self.__class__ == other.__class__ and self._value == other._value
def __ne__( self, other ):
    return self.__class__ != other.__class__ or self._value != other._value
```

We make a token effort at making a cell more-or-less immutable. This makes it hashable.

```
@property
def value( self ):
    return self._value
```

Todo

Test hashable interface of Cell

1.4.6 EmptyCell

class `cell.`**`EmptyCell`**

An `EmptyCell` implements empty cells. `xlrd` may report them as a type `XL_CELL_EMPTY`. A Numbers spreadsheet may use the <o> or <g> tag.

```
class EmptyCell( Cell ):
    """The *value* will be ’’, but we ignore that."""
    def is_empty( self ): return True
    def to_int( self ): return None
    def to_float( self ): return None
    def to_decimal( self, digits=None ): return None
    def to_str( self ): return None
    def to_datetime( self, format=None ): return None
    def to_digit_str( self, len=None ): return None
```

1.4.7 TextCell

class `cell.`**`TextCell`**

A `TextCell` implements the cells with text values. `xlrd` may report them as a type `XL_CELL_TEXT`. It's often possible to interpret the text as some other value, so the conversions make reasonable attempts at that.

This is used for CSV workbooks as well as XLS workbooks. This is the default type for Fixed format files, also.

Note that COBOL files will explicitly have bytes values, not string values.

```
class TextCell( Cell ):
    """A Cell which contains a Python string value."""
    def to_int( self ):
        return int( self.value )
    def to_float( self ):
        return float( self.value )
    def to_decimal( self, digits=0 ):
        return decimal.Decimal( self.value )
    def to_str( self ):
        return self.value
    def to_datetime( self, format=None ):
        if format is None:
            try:
                format = locale.nl_langinfo(locale.D_FMT)
            except AttributeError as e:
                # Windows
                format = "%x"
        return datetime.datetime.strptime(self.value,format)
    def to_digit_str( self, length=5 ):
        fmt= "{{0:0>{0}d}}".format(length)
        return fmt.format( int(self.value) )
```

1.4.8 NumberCell

class `cell.`**`NumberCell`**

A `NumberCell` implements the cells with a float value. `xlrd` may report them as a type `XL_CELL_NUMBER`. A variety of conversions make sense for a number value. Note that the `to_datetime()` conversion depends on `xlrd`. This may be a faulty assumption for some species of workbooks.

```
class NumberCell( Cell ):
    """A cell which contains a Python float value."""
    def to_int( self ):
        return int( self.value )
    def to_float( self ):
        if isinstance(self.value,float):
            return self.value
        # likely, it's Decimal!
        return float(self.value)
    def to_decimal( self, digits=0 ):
        if isinstance(self.value,float):
            fmt= "{0:0.{digits}f}"
            return decimal.Decimal( fmt.format(self.value, digits=digits) )
        elif isinstance(self.value,decimal.Decimal):
            return self.value
        else:
            return decimal.Decimal(self.value)
    def to_str( self ):
        return str(self.value)
    def to_datetime( self, format=None ):
        assert format is None, "Format is not used."
        try:
            dt= xlrd.xldate_as_tuple(self.value, self.workbook.datemode)
        except xlrd.xldate.XLDateAmbiguous as e:
            ex= ValueError( "Ambiguous Date: {0}".format(self.value) )
            ex.__cause__= e
            raise ex
        return datetime.datetime(*dt)
    def to_digit_str( self, length=5 ):
        fmt= "{{0:0>{0}d}}".format(length)
        return fmt.format( int(self.value) )
```

1.4.9 FloatDateCell

class cell.**FloatDateCell**

A FloatDateCell implements the cells with XL_CELL_DATE. Since the conversions are all identical to number, we simply inherit the features of a number.

```
class FloatDateCell( NumberCell ):
    """A cell which contains a float value that is actually an Excel date."""
    pass
```

Other formats have other kinds of date cells that aren't simply dressed-up floating-point numbers.

1.4.10 BooleanCell

class cell.**BooleanCell**

A BooleanCell implements the cells with XL_CELL_BOOLEAN. Since the conversions are all identical to number, we simply inherit the features.

```
class BooleanCell( NumberCell ):
    """A cell which contains a boolean value."""
    pass
```

1.4.11 ErrorCell

class cell.**ErrorCell**

An ErrorCell implements the cells with XL_CELL_ERROR. The only sensible conversion is ErrorCell.to_str() which reports the error string for the cell.

```
class ErrorCell( Cell ):
    """A cell which contains an error code."""
    def to_int( self ):
        raise ValueError( self.value )
    def to_float( self ):
        raise ValueError( self.value )
    def to_decimal( self, digits=0 ):
        raise ValueError( self.value )
    def to_str( self ):
        return self.value
    def to_datetime( self, format=None ):
        raise ValueError( self.value )
    def to_digit_str( self, length=5 ):
        raise ValueError( self.value )
```

1.4.12 DateCell

class cell.**DateCell**

A `DateCell` implements a cell with a proper date-time value. This is a value which did not come from a workbook float value.

This could be a parsed string, for example.

A variety of conversions make sense for a proper date value.

```
class DateCell( Cell ):
    """A cell which contains a proper :mod:`datetime` value."""
    def to_int( self ):
        return int(self.to_float())
    def to_float( self ):
        timetuple= self.value.timetuple()[:6]
        xl= xlrd.xldate.xldate_from_datetime_tuple(
            timetuple,
            self.workbook.datemode)
        return xl
    def to_decimal( self, digits=0 ):
        fmt= "{0:0.{digits}f}"
        return decimal.Decimal( fmt.format(self.to_float(),digits=digits) )
    def to_str( self ):
        return str(self.value)
    def to_datetime( self, format=None ):
        return self.value
    def to_digit_str( self, length=5 ):
        fmt= "{{0:0>{0}d}}".format(length)
        return fmt.format( self.to_int() )
```

For numbers, the `<d>` cells have a native date format.

This is an extension to basic `xlrd`, XLSX and ODS workbook processing because no cell has this as its data type.

1.4.13 Conversion Functions

There is a need for functions for various kinds of conversions. These are mostly focused on processing Fixed format files, where the data all originates as text.

These can also be applied to CSV or TAB files to create cells other than the default `cell.TextCell`.

Dates

The function `date_from_string()` is a closure based on a format string that returns a single-argument conversion function.

`cell.`**`date_from_string`**(*format*)

```
def date_from_string( format ):
    def the_conversion( string ):
        return datetime.datetime.strptime( string, format )
    return the_conversion
```

This forms the factory for the `cell.DateCell` class.

`cell.`**`datecell_from_string`**(*format*)

```
def datecell_from_string( format ):
    dt_conv= date_from_string( format )
    def the_conversion( string, workbook ):
        return DateCell( dt_conv( string ), workbook )
    return the_conversion
```

This could be used like this in a schema definition.

```
d = Attribute( name="mm-dd-yy", size=*n*, offset=*m*,
    create=stingray.cell.datecell_from_string("%m/%d/%y") )
```

The function `date_from_float()` is an additional closure based on a `workbook.datemode` that returns a single-argument conversion function.

`cell.`**`date_from_float`**(*format*)

```
def date_from_float( datemode ):
    def the_conversion( value ):
        try:
            dt= xlrd.xldate_as_tuple(value, datemode)
        except xlrd.xldate.XLDateAmbiguous as e:
            ex= ValueError( "Ambiguous Date: {0!r}".format(value) )
            ex.__cause__= e
            raise ex
        return datetime.datetime(*dt)
    return the_conversion
```

This function has similar applications for converting data to a more useful `cell.DateCell` instance.

```
float2date= stingray.cell.date_from_float(workbook.datemode)
d = Attribute( name="mm-dd-yy", size=*n*, offset=*m*,
    create=lambda x, w: stingray.cell.DateCell( float2date(x), w ) )
```

Numbers

A fixed-format file can have any of a variety of numbers, encoded in a variety of ways.

The ordinary number-as-string, is trivially handled by `cell.TextCell`. No real need for a more sophisticated conversion.

The less-ordinary case of COBOL data, in computational-3 format, or display format with an assumed decimal place is more difficult. We'll defer this to *The COBOL Package*.

1.5 Sheet Module – Sheet and Row Access

A *Sheet* is a generator of *Row* objects. A *Row* is a sequence of `cell.Cell` instances, identified by position.

We have three variations on `sheet.Sheet`.

- A simple `sheet.Sheet` lacks a schema. (This corresponds with `csv.reader()`.) For workbooks with a well-known physical format, the schema can be optional. Each `sheet.Row` object can be built eagerly and accessed by position.
- A sheet with a schema. There are two variations.
  – `sheet.EmbeddedSchemaSheet` contains a schema. This could be a simple as column titles in the first row. (This corresponds to `csv.DictReader`.) Or it could be considerably more complex.
  – `sheet.ExternalSchemaSheet` requires an external schema. This schema may be simply a list of column titles supplied externally. More often, the schema is a complete physical format description for Fixed or COBOL format files.

A known physical format (like a workbook) can build sheet.Row objects eagerly with or without a schema.

In the case of COBOL and fixed-format files, however, a sheet.Row cannot be built eagerly. It must be a lazy object which only builds cell.Cell as needed. See *The COBOL Package* for details.

1.5.1 Get Embedded Schema Use Case

For an sheet.EmbeddedSchemaSheet, the application (or Workbook) must do a three-step dance to get the schema that is embedded in the sheet.

1. Build a sheet.EmbeddedSchemaSheet with an "embedded schema loader" class. (For example, schema.loader.HeadingRowSchemaLoader.) The loader partitions rows into two sets: header and data.
2. Load the schema from the sheet. The sheet.Sheet will build an object of the loader class and use it to gather the schema information. The schema loading may involve skipping irrelevant rows or combining multi-line headings or anything else required to parse the sheet.
3. Get the rows from the sheet. This will, also, invoke the attached loader to filter rows so that the header is not seen as data.

```
with open as wb:
    sheet = EmbeddedSchemaSheet( workbook, 'Sheet1', HeadingRowSchemaLoader )
    counts= process_sheet( sheet )
    pprint.pprint( counts )
```

1.5.2 Get External Schema Use Case

For an sheet.ExternalSchemaSheet, the application (or Workbook) must do a four-step dance to get the schema.

1. Build a schema loader. This loader will require a source workbook, sheet name and a reader object.
2. Get the Schema object from the loader.
3. Build a sheet.ExternalSchemaSheet with the Schema object.
4. Get the rows from the sheet.

And yes, the external source, is another spreadsheet! Worse, the external source could be a fixed file or workbook for which a meta-schema is required to read the schema.

```
with open schema as swb:
    esl = ExternalSchemaLoader( swb, sheet_name='Schema' )
    schema = esl.load()
with open data as wb:
    sheet = ExternalSchemaSheet( wb, 'Sheet1', schema )
    counts= process_sheet( sheet )
    pprint.pprint( counts )
```

1.5.3 Get Rows Use Case

The essential job of a sheet.Sheet is to produce sheet.Row instances. A row is a sequence of cell.Cell instances.

Note that csv is eager about building a row from the source data. This isn't universally appropriate. COBOL files require lazy construction of the row's cells.

A schema.Schema can transform a sequence row into a dictionary row or a named tuple row. The schema.Attribute.name becomes the key for this row-as-dictionary.

We specifically delegate the row-as-dictionary interpretation to the schema.Schema, and avoid doing it in the sheet.Sheet. This is because most workbook schemata are flat. However, a COBOL schema can have a very complex structure, making the row-as-dictionary too simplistic to be useful.

As noted above, there are two candidate implementations of a Row.

- **Eager**. Appropriate for most (but not all) Physical Formats. The idea is to apply the schema immediately to create the row as a tuple of cells. csv does this, and it can be applied to other workbook formats. It can be applied to simple, flat Fixed format files.
- **Lazy**. This is more appropriate for Fixed format files and COBOL format files. Specifically, the data conversion, redefines and repeating group issues force us to wait for cell access rather than immediately create all possible cells. Indeed, for COBOL files with REDEFINES definitions, some of the cells cannot be built eagerly; application logic must determine which attributes are valid or invalid.

Note that the API is the same. The implementation differs.

Here's our prototypical code.

```
def process_sheet( sheet ):
    counts= defaultdict( int )
    for row in sheet.rows():
        #row is a sequence of Cell instances
        print( repr(c) for c in row )
        counts['read'] += 1
    return counts
```

Ultimately, the sequence nature of a row is unsatisfying. We'll have to wait until *Schema Package – Schema and Attribute Definitions* to extend this into something useful.

1.5.4 Sheet Identification

For CSV and TAB files, as well as COBOL and Flat files, there is one anonymous "sheet" that is the entire workbook.

For XLS, XLSX, and ODS formats, however, there are sheets within the workbook.

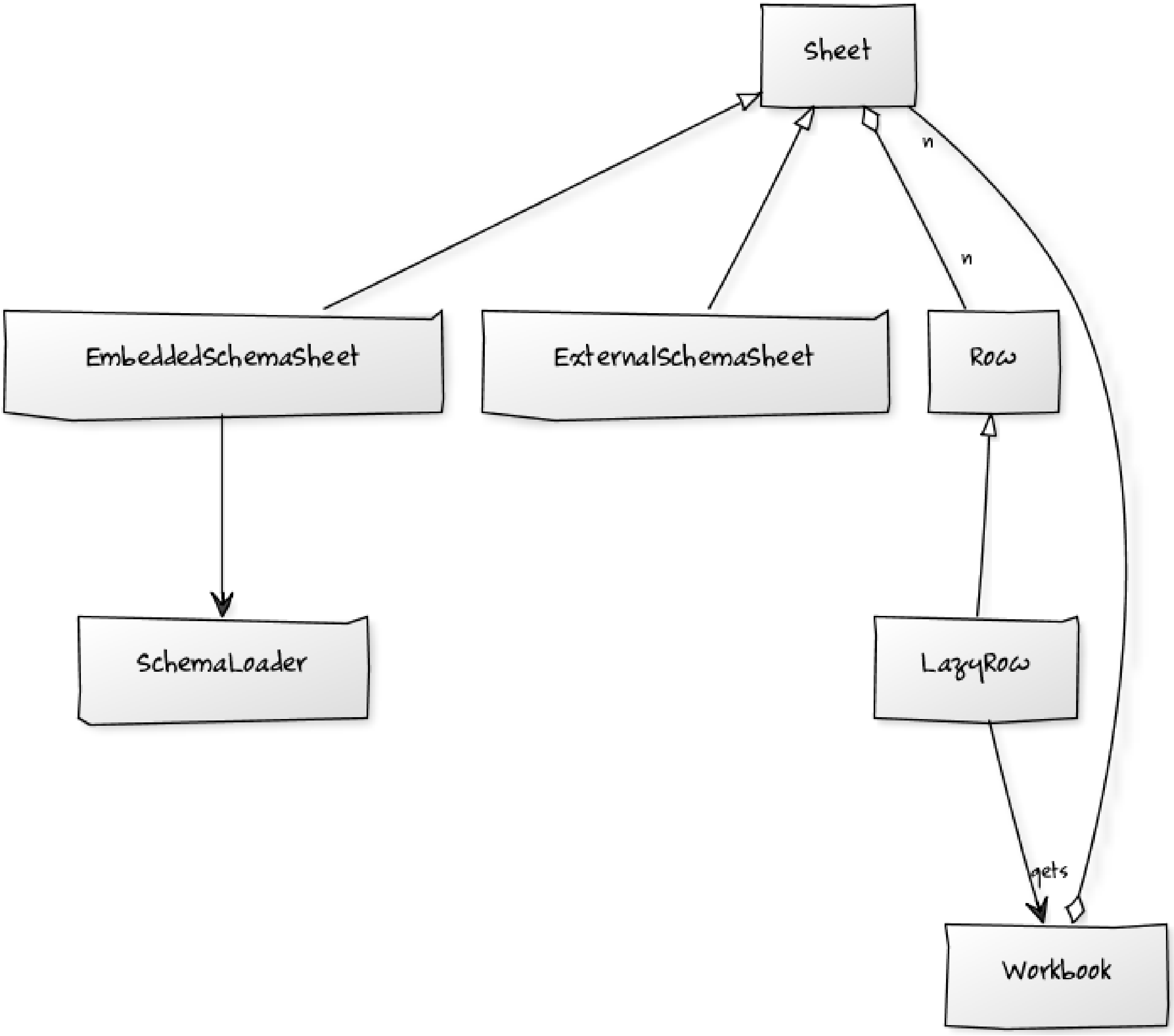
For Numbers, there are "pages" or "workspaces" that have multiple tables. Each Numbers **table** is – effectively – a sheet.Sheet. The intermediate organization level, "workspace", is an additional detail.

We handle this in the following way.

- One anonymous sheet has a name either of None or the basename of the file.
- Simple sheets have names which are simple strings.
- Numbers workspaces with sheets have names which are two-tuples of workspace ("sheet") and table name.

1.5.5 Model

```
http://yuml.me/diagram/scruffy;dir:td/class/
#sheet,
[Workbook]<>-n[Sheet],
[Sheet]<>-n[Row],
[Row]^[LazyRow],
[LazyRow]-gets->[Workbook],
[Sheet]^[EmbeddedSchemaSheet],
[Sheet]^[ExternalSchemaSheet],
[EmbeddedSchemaSheet]->[SchemaLoader].
```



1.5.6 Overheads

Sheet and Row are essentially lazy sequences.

```
"""stingray.sheet -- Defines Row as  a collection of Cells and Sheet as a collection of Rows.
"""
from collections import Sequence
```

There are two "implicit" dependencies, also. A row depends on details of an `schema.Attribute` and a `workbook.Workbook`. However, there's no real need to present a formal import for this. The Attribute and Workbook are simply opaque objects passed around as arguments.

1.5.7 Sheet Class

class `sheet.`**`Sheet`**

A `sheet.Sheet` is an iterator over the rows of data in a workbook. Subclasses implement different bindings for the sheet's schema information.

```
class Sheet:
    """An iterator over rows.
       A binding to a workbook.
       A subclass of Sheet will be bound to a schema.
    """
    def __init__( self, workbook, sheet_name ):
        self.workbook, self.name= workbook, sheet_name
    def __repr__( self ):
        return "{0}({1!r},{2!r})".format( self.__class__.__qualname__,
            self.workbook, self.name )
    def rows( self ):
        """Iterate through the rows of this sheet.
        This is a convenient interface for ''self.workbook.rows_of(self)''
        """
        return self.workbook.rows_of( self )
```

1.5.8 Row Class

class `sheet.`**`Row`**

A single row in Sheet; a sequence of `cell.Cell` instances.

A Sheet produces this simple row-as-list. A Schema can transform this into row-as-dict or some even more elaborate structure.

A row depends on details of an `schema.Attribute` and a `workbook.Workbook`. This feels circular, but doesn't present any real problems.

The `cell.Cell` conversions are handled by the Workbook. Some Workbooks have cell content identified by position. Some Workbooks have cell content identified by size, offset and encoding. Therefore, we must provide the Attribute details to the Workbook to get the Cell's value.

```
class Row( Sequence ):
    """Eager Row: a tuple of Cell values."""
    def __init__( self, sheet, *data ):
        """Build another Row.

        :param sheet: the containing sheet.
        :param *data: the various Cell values in this row
        """
        self.sheet= sheet
        self.data= data
    def cell( self, attribute ):
        """Get a specific cell, based on a schema Attribute.

        :param attribute: The attribute's value to return.
        """
        return self.sheet.workbook.row_get( self, attribute )
    def __len__( self ):
        return len(self.data)
    def __iter__( self ):
        return iter(self.data)
    def __contains__( self, cell ):
        return any( cell.value == d.value for d in self.data )
    def __getitem__( self, index ):
        return self.data[index]
```

To approach the csv.DictReader API (without the eager processing), we need make the Row API slightly more fluent with a by_name() method.

```
def by_name( self, name ):
    attr= self.sheet.schema.get_name(name)
    return self.cell( attr )
```

Note that the presumption in this interface is that the Attribute is sufficiently detailed to specify a single cell.Cell. For non-COBOL workbooks, this is perfectly true.

For COBOL, however, there are groups and occurs clauses, meaning that a single Attribute can represent multiple cell.Cell instances. Which one do we mean? And how do we specify this selection?

- The sheet.Row.cell() method can return a structure with all the values. Ordinary Python can then pick apart the instances. This requires working up the DDE hierarchy to locate all of the applicable "occurs" by to construct the proper dimensionality of an attribute.

  It also means getting all of the values to create a tuple or nested tuple-of-tuple structure for the various dimensions.

- The schema.Attribute.index method selects data from the row in the workbook. This applies the indices to the Attribute to compute the required offset into the source data.

class sheet.**LazyRow**

When we can't eagerly build all cell.Cell instances for a given row, this class provides the proper API.

A COBOL REDEFINES clause may make the bytes invalid in all but one of the aliases for an attribute. Also, there's no formal NULL value in COBOL, so optional fields can have invalid data.

Further, we may have Occurs Depending On. This means we can't set size and offset until we can access actual data.

For these reasons, we have a sheet.LazyRow, which conforms to the interface for a Row, but isn't an actual sequence. No data is processed until the LazyRow.__getitem__() method is used.

```
class LazyRow( Sequence ):
    """Lazy Row: a tuple-like sequence of Cell values."""
    def __init__( self, sheet, **state ):
        """Build another Row.

        :param sheet: the containing sheet.
        :param **state: worksheet-specific state value to save.
        """
        self.sheet= sheet
        self._state= state
        super().__init__()
    def __repr__( self ):
        return "LazyRow(sheet={0!r}, state={1!r})".format( self.sheet, self._state )
    def cell( self, attribute ):
        """Get a specific cell, based on a schema Attribute.

        :param attribute: The attribute's value to return.
        """
        return self.sheet.workbook.row_get( self, attribute )
    def __len__( self ):
        return len(self.sheet.schema)
    def __iter__( self ):
        for attribute in self.sheet.schema:
            try:
                yield self.sheet.workbook.row_get( self, attribute )
            except Exception as e:
                yield None
    def __contains__( self, cell ):
        for attribute in self.sheet.schema:
            try:
                col= self.sheet.workbook.row_get( self, attribute )
            except Exception as e:
                pass
            if col.value == cell.value:
                return True
    def __getitem__( self, index ):
        attribute= self.sheet.schema[index]
        return self.sheet.workbook.row_get( self, attribute )
```

To approach the `csv.DictReader` API (without the eager processing), we can make the `Row` API slightly more fluent with a `by_name()` method.

```
def by_name( self, name ):
    attr= self.sheet.schema.get_name(name)
    return self.cell( attr )
```

1.5.9 ExternalSchemaSheet Class

class `sheet.`**`ExternalSchemaSheet`**

A Sheet with an external schema can be one of two kinds.

- A Sheet that doesn't have row headers to embed the schema information. In this case, an eager Workbook Row can create a sequence of `cell.Cell` instances. The Schema information can be associated by position.
- A Sheet that is really a COBOL or Fixed format file. In this case, the Workbook cannot create a sequence of `cell.Cell` instances. Instead, the Sheet (which has schema information) must provide a LazyRow with deferred Cell conversions.

```
class ExternalSchemaSheet( Sheet ):
    """A Sheet with an external Schema."""
    def __init__( self, workbook, sheet_name, schema ):
        """Initialize a sheet for processing.

        :param workbook: the containing workbook
        :param sheet_name: the specific sheet to locate within the Workbook
        :param schema: the :py:class:'schema.Schema' schema definition.
        """
        super().__init__( workbook, sheet_name )
        self.schema= schema
    def rows( self ):
        """Iterate through the rows of this sheet."""
        return self.workbook.rows_of( self )
```

1.5.10 EmbeddedSchemaSheet Class

class `sheet.`**`EmbeddedSchemaSheet`**

A sheet with an embedded schema must have a loader class provided. The loader is invoked to build a `schema.Schema` object. It's also used to return the rest of the rows; those that weren't used to build the schema.

```
class EmbeddedSchemaSheet( ExternalSchemaSheet ):
    """A Sheet with a Schema embedded in it."""
    def __init__( self, workbook, sheet_name, loader_class ):
        """Initialize a sheet for processing.

        :param workbook: the containing workbook
        :param sheet_name: the specific sheet to locate within the Workbook
        :param loader_class: the :py:class:`schema.loader.SchemaLoader`
        schema loader to load the schema from the sheet.

        Apply the loader to the given sheet of the workbook to get schema
        and rows.
        """
        s = Sheet( workbook, sheet_name )
        self.loader = loader_class( s )
        schema= self.loader.schema()
        super().__init__( workbook, sheet_name, schema=schema )
    def rows( self ):
        """The parser will skip over the headers."""
        return self.loader.rows()
```

Since the rows are already properly encoded as `cell.Cell` instances, no further processing is required by the Sheet or the Loader.

1.5.11 Rows of a Sheet

Note that the `csv` design pattern for each row involves two subclasses with the same method names but different results. One returns a `dict` of cells, the other returns a `list` of cells.

The dict-based processing has the advantage of clarity. It has the disadvantage of not coping well with duplicate column names or data which breaks first normal form.

Also, note that `csv` does eager creation of each row. The `csv.DictReader` does eager creation of a dictionary from each row.

We don't follow the `csv` design pattern. Instead we do the following.

- A `sheet.Row` can be a lazy sequence of `cell.Cell` instances.
- A `schema.Schema` must be used to fetch `cell.Cell` instances from the `sheet.Row`.
- To create dict-like access to Cell instances, the schema can be turned into a dictionary. This "schema-as-dict" can then be used with a properly lazy Row to create Cell instances.

This lazy evaluation of a row that fetches data based on `schema.Attribute` details allows us to cope with COBOL `REDEFINES`. It also allows us to cope with the unfortunately common problem of duplicate column names in conventional spreadsheets.

We can have application programming which looks like this to process rows in a number of ways.

Row as sequence is the default.

```
for row in sheet.rows():
    Cell: row[i]
    Schema Attribute Name: sheet.schema[i].name
```

Row as dict is a common alternative. If we have unique column names in the schema, We can than use application programming that looks like this.

```
schema_dict = dict( (a.name, a) for a in sheet.schema )
for row in sheet.rows():
    Cell: row.cell(schema_dict['name'])
    row_as_dict= dict(
        (a.name, row.cell(a)) for a in sheet.schema )
    Cell: row_as_dict['name']
```

This handles the COBOL case, where rows must be lazy. This includes COBOL `REDEFINES` and occurs clauses. This assures proper packed decimal conversion of redefined fields.

1.6 Schema Package – Schema and Attribute Definitions

A *Schema* contains data descriptions. It's a collection of *Attribute* specifications.

For our purposes, we're trying to cover the following bases:

- Workbooks in a variety of forms: CSV, Tab, XLSX, ODS, etc. The common feature of these schema is that they're flat. A single index (or name) identifies each column.
  - The schema may be a header row within a sheet.
  - The schema may be elsewhere in the sheet. It may be a header that's not the first row.
  - The schema may be in a separate document. It could be XSD, but this is vanishingly rare.
- COBOL Files. The schema is a separate document, encoded in COBOL source code. A COBOL schema is rarely a perfectly flat structure. However, using `.`-path names allows us to flatten a COBOL structure into a simpler structure that can be compatible with workbooks.

We'll also happen to cover relational database table definitions. However, this isn't our focus. This is simply coincidence.

Note that we are not trying to cover XML schemas or a complete relational database schema.

1.6.1 Load a Schema Use Case

The objective of this module is to provide a handy base class family that can be used to load schema information from any of a variety of sources.

- Embedded schema in a sheet of a workbook. Either a header row or some more complex parsing. This implies some kind of schema parser that reads a workbook sheet and gets little more than column names and ordinal positions.
- External schema in another workbook. This implies a parser that gathers some information (name, type, position, offset, or size) from a sheet using a fixed schema.
- External schema in COBOL. This implies a parser for COBOL source.

Embedded schema loading would look like this.

```
with open as wb:
    counts= defauldict( int )
    sheet = EmbeddedSchemaSheet( workbook, 'Sheet1', HeadingRowSchemaLoader )
    counts= process_sheet( sheet )
    pprint.pprint( counts )
```

The schema object isn't made explicit. It's available in `sheet.schema`.

External schema loading would look like this.

```
with open schema as swb:
    esl = ExternalSchemaLoader( swb, 'Schema' )
    schema = esl.load()
with open data as wb:
    sheet = ExternalSchemaSheet( wb, 'Sheet1', schema )
    counts= process_sheet( sheet )
    pprint.pprint( counts )
```

1.6.2 Use a Schema Use Case

We use a schema to access fields by name. Here's the use case from *Introduction*.

```
foo= row['foo'].to_str()

bar= row['bar'].to_float()
```

This, of course, only works in the cases where the field name is unique and the row's values can be built eagerly. The good news is that 80% of the time this is true. The other 20% of the time, we need something more complex.

For the 80% case, we can do this.

```
for row_seq in sheet.rows():
    row= dict(
        (a.name, row_seq.cell(a))
        for a in schema
    )
    foo= row['foo'].to_str()
    bar= row['bar'].to_float()
```

Or this.

```
for row in schema.rows_as_dict_iter( sheet.rows() ):
    foo= row['foo'].to_str()
    bar= row['bar'].to_float()
```

In the 20% case, we can't build a row eagerly. In this case, we have to do the following to fetch cell values using a properly lazy row.

```
schema_dict= dict( (a.name, a) for a in schema )
for row_seq in sheet.rows():
    foo= row.cell( schema_dict['foo'] ).to_str()
    bar= row.cell( schema_dict['bar'] ).to_float()
```

This involves a three-step dance because a row (and a schema) have a number of ambiguities. In particular, names may be duplicated, forcing us to use position or more complex naming conventions. Also, the attribute may have repeating groups, requiring some indexing, as well as naming.

1. Build a schema mapping by some name. We can use the attribute name (if it's unique) or (for COBOL schema) a unique path name. Or, we may have some other, more complex naming for attributes. Really.
2. Find the attribute in our mapping.
3. Find the cell value based on the attribute.

1.6.3 The Builder

Most applications are based on building Python objects from source file data. Experience indicates that the conceptual schema may have several variant logical layout implementations. Because of the logical layout variability, a two-step dance is required between source rows and final Python objects.

1. Transform input row to a standardized dictionary. The mapping from input layout to dictionary changes. Frequently. The dictionary matches the conceptual schema. The input is one of the variant logical layouts.

2. Create the application object from the standardized dictionary.

```
def build_dict_1( aRow ):
    return dict(
        attribute= aRow['column'].to_str(),
        another= aRow['data'].to_float(),
    )

def make_app_object( aDict ):
    return Object( **aDict )

def process_sheet( sheet, builder=build_dict_1 ):
    counts= defaultdict( int )
    object_iter = (
        make_app_object(builder(row))
        for row in sheet.schema.rows_as_dict_iter(sheet.rows()) )
    for obj in object_iter:
        process object
    return counts
```

This allows us to configure an appropriate builder function depending on which variation of the logical layout the file actually has.

The `csv` module offers trivial support for an eager "row-as-dict" processing. This can actually introduce problems. Specifically, COBOL record layouts may have a number of fields named `Filler`. Really. Also, it's common to get data where column names are duplicated in the embedded schema. Finally COBOL redefines means that lazy construction of Cell instances is more appropriate.

Data conversions can become a $n \times m$ issue. Each of n input types can be mapped to each of m output types. In this instance, we try to keep it to $n + m$. We do this by acquiring a Python-specific type from the source.

- For fixed format files, this may involve decoding characters (not bytes).
- For COBOL format files, this may involve decoding bytes if EBCDIC and COMP-3 data are involved. It may simply involve decoding characters if the file happens to be encoded into proper characters.
- For all other physical formats (CSV, XLS, XSLX, etc.) there is no separate decoding. There are about five canonical cell types (mostly floats) with decodings defined by the format.

Byte to Character conversion is not part of the schema problem. That's part of the physical format. For the most part, the physical format defines the encoding of the bytes. COBOL files in Unicode (or ASCII) strings, require standard, default decoding. COBOL files in EBCDIC, however, may require highly customized decoding. This will be delegated to that module.

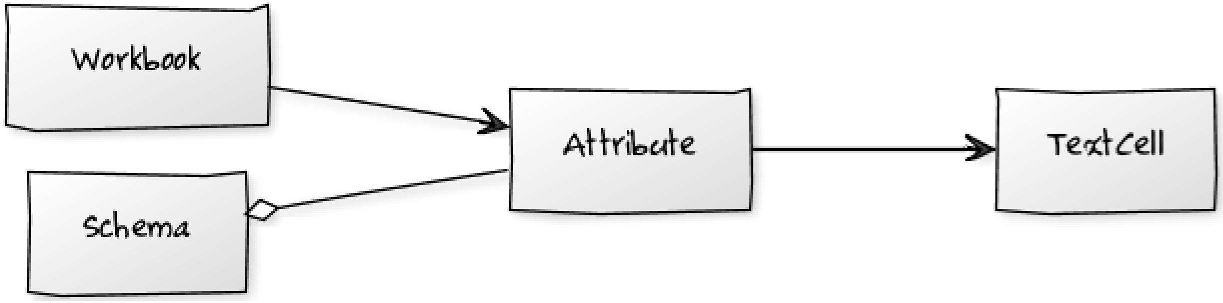
Output conversions are not part of the schema problem. They're part of the application. All this module does is get a workbook `cell.Cell` instance.

It leads us builder functions that might look like this.

```
def build_dict_2( aRow ):
    if aRow['flag'].to_str() == "C":
        value= math.pi*aRow['r'].to_float()**2
    else:
        value= aRow['l'].to_float()*aRow['w'].to_float()
    return dict(
        attribute= aRow['column'].to_str(),
        another= aRow['data'].to_float(),
        value= value,
    )
```

1.6.4 Model

```
http://yuml.me/diagram/scruffy;/class/
#schema,
[Schema]<>-[Attribute],
[Attribute]->[TextCell],
[Workbook]->[Attribute].
```



1.6.5 Overheads

A schema depends only on the definitions in `cell`.

```
"""stingray.schema -- Defines an overall Schema of Attributes which can be
embedded as a row of a sheet or encoded in a separate sheet.  A
schema defines the logical layout of a workbook or flat file.
"""

import datetime
import time
import stingray.cell
```

1.6.6 Schema Class

class `schema.`**`Schema`**

The core Schema definition is an extension to `list`. In addition to a sequence of attributes, it also has an "info" object that's a dictionary of additional keywords.

The `schema.Schema.rows_as_dict_iter()` method uses the sheet's `sheet.Sheet.rows()` iterator to create simple row-as-list values. These are transformed into the row-as-dict values. If the attribute names involve duplicates, then one of the duplicated values will be chosen; the choice is arbitrary.

info Dict of additional information about this schema. Meta-metadata. For COBOL schema, this includes the source DDE.

names Attribute names for rows_as_dict_iter()

```
class Schema( list ):
    """A Mutable Sequence of attributes.  Order matters.
    """
    def __init__( self, *attr, **kw ):
        """Build a schema from collection of attributes."""
        super().__init__( attr )
        for p, a in enumerate( self ):
            a.position= p
        self.info= kw
    def __repr__( self ):
        attr_list= map( repr, self )
        return "Schema( {0} )".format( ", ".join(attr_list) )
    def rows_as_dict_iter( self, sheet ):
        self.names= tuple(a.name for a in self)
        for r in sheet.rows():
            yield dict(
                (a.name, r.cell(a)) for a in sheet.schema )
    def append( self, child ):
        child.position= len(self)
        super().append( child )
```

Possibly helpful method to expand a row based on the schema information.

```
def expand( self, aRow ):
    """Expand each attribute to create a dictionary of cells."""
    return dict( (attr.name, aRow.cell(attr)) for attr in self )
```

For parsing COBOL data, we often need to know the total length of the defined schema. This only works for records without an Occurs Depending On.

```
def lrecl( self ):
    return max( a.offset + a.size for a in self )
```

A Schema needs to handle two common use cases.

- Most formats. The items are defined by the physical format. Data can be fetched positionally. Names map to positions.
- Fixed and COBOL. The columns are not defined by the physical format, but by an external schema associated with the `sheet.Sheet`. Names map to offsets and sizes; these must be computed from the external schema. In the case of Occurs Depending On (ODO), the offsets depend on both schema and data.

  COBOL data may have elements which are invalid, but unused due to application logic in selecting a proper REDEFINES alias.

The simple positional schema isn't really appropriate for all purposes. For COBOL and fixed format files with external schema, we often must process things lazily by field name.

This is unlike spreadsheets where we can process all fields eagerly and in order.

Todo

Index by name and path, also.

This will eliminate some complexity in COBOL schema handling where we create the a "schema dictionary" using simple names and path names.

1.6.7 Attribute Class

class `schema.`**`Attribute`**

An Attribute definition has a required value of a name and a class that will be created to hold the data.

Here are the essential attributes of an Attribute.

name The attribute name. Typically always available for most kinds of schema.

create Cell class to create. If omitted, the class-level `Attribute.default_cell` will be used. By default, this refers to `stingray.cell.TextCell`.

The additional values commonly provided by simple fixed format file schemata.

offset Optional offset into a buffer. For simple fixed-layout files, this is a constant. For COBOL files with Occurs Depending On, however, this must be a function based on the actual record being processed.

size Optional size within the buffer.

position Optional sequential position.

A subclass might introduce yet more attributes.

```
class Attribute:
    """Essential definition of a single source data element."""
    default_cell= stingray.cell.TextCell
    def __init__(self, name, offset=None, size=None, create=None, position=None, **kw):
        """Build an Attribute.
        :param name: The attribute name.
        :param offset: Optional offset into a buffer.
        :param size: Optional size within the buffer.
        :param create: Cell class to create.  If omitted, the class-level
            :py:data:`Attribute.default_cell` will be used.
        :param position: Optional sequential position.
        """
        self.name, self.offset, self.size, self.create, self.position =\
        name, offset, size, create, position
        if not self.create:
            self.create= self.default_cell
        self.__dict__.update( kw )
    def __repr__( self ):
        return "Attribute( name={0.name!r}, position={0.position}, offset={0.offset}, size={0.size} )
```

An `schema.Attribute` is used by a `workbook.Workbook` to extract cell data from a row.

The use case looks like this for a Fixed format workbook. For other workbooks, other kinds of conversion functions might be used.

```
def cell( sheet, attribute, data ):
    a= attribute
    return a.create( data[a.offset:a.offset+a.size], sheet.workbook )
```

The attribute might be declared as follows.

```
Attribute( name= ``mm-dd-yy'', size= n, offset= m,
    create=SomeCellSubclass )
```

1.7 Schema Loader Module – Load Embedded or External Schema

A *Schema Loader* loads the attributes of a schema from a source document. There are a variety of sources.

- The first row of a sheet within a workbook. This version has to be injected into workbook processing so that the first row is separated from the data rows.
- A separate sheet of a workbook. This version requires a sheet name.
- A separate workbook. This, too, requires a named sheet.
- COBOL Code. We'll set this aside as a subclass so complex it requires it's own module.

A schema loader is paired with a specific kind of `sheet.Sheet`.

A workbook requires a schema, which requires a schema loader. A schema loader depends on a meta-workbook. Ideally that meta-workbook has an emedded schema, but it may have an external schema, meaning we could have a meta-schema required load the schema for the application data. Sheesh.

First, let's hope that doesn't happen. Second, the circularity is resolved by making it the responsibility of the the application to handle schema loading.

1.7.1 Embedded Schema Use Case

A `sheet.EmbeddedSchemaSheet` requires a loader class. The loader will

1. Be built with the sheet as an argument.
2. Be interrogated for the schema.
3. Be interrogated for the rows.

The most typical case is the single-header-row case.

In some cases, the loader is actually a a rather sophisticated parser that paritions the data into the embedded schema and the data rows.

```
with Workbook( name ) as wb:
    sheet = self.wb.sheet( ‘Sheet2’,
        stingray.sheet.EmbeddedSchemaSheet,
        loader_class= stingray.schema.loader.HeadingRowSchemaLoader )

    for row in sheet.rows():
        process the row
```

1.7.2 External Schema Use Case

A `sheet.ExternalSchemaSheet` requires a schema.

In the typical case, the external schema file has an emedded meta-schema. The first row has appropriate column names. This requires a subclass of `schema.loader.ExternalSchemaLoader` to properly map the names that were found onto the attributes of the `schema.Attribute` class.

When the embedded meta-schema has unusual names, then a builder must be defined to map the names that are found in the schema and build an `schema.Attribute` instance.

```
with open_workbook( schema_name ) as schema_wb:
    esl= stingray.schema.loader.ExternalSchemaLoader( schema_wb, "Schema" )
    schema= esl.schema()
with Workbook( name, schema=schema ) as wb:
    sheet = self.wb.sheet( ’Sheet2’,
        stingray.sheet.ExternalSchemaSheet,
        schema= schema )
    counts= process_sheet( sheet )
    pprint.pprint( counts )
```

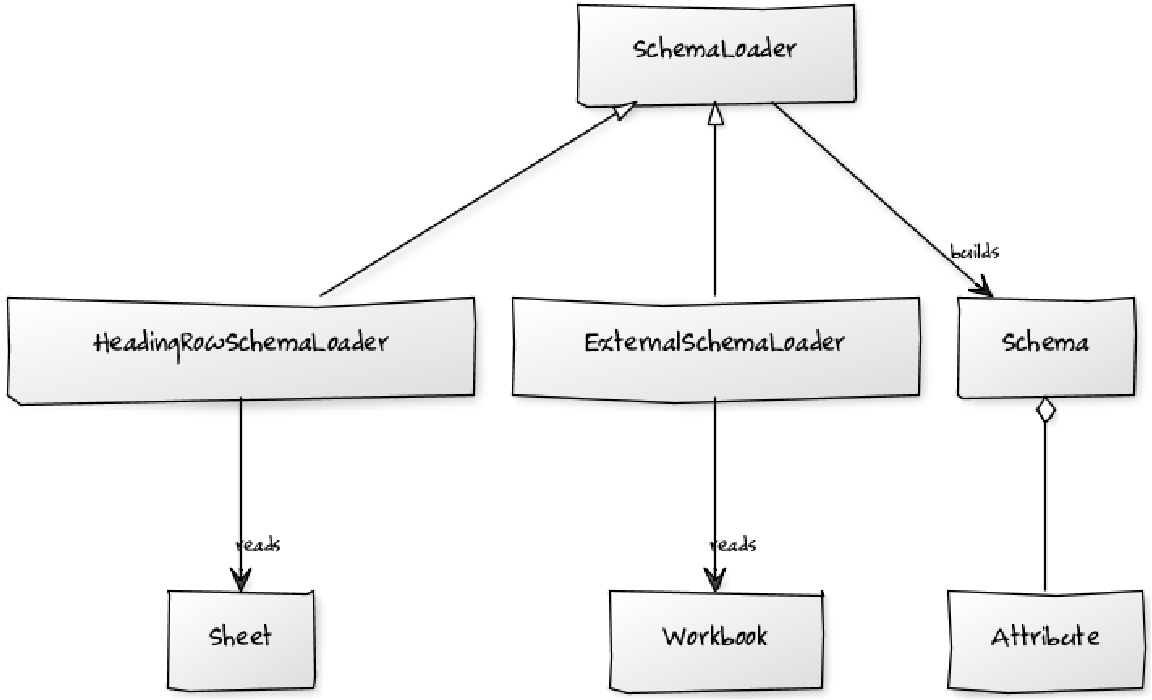
1.7.3 Manual Schema Use Case

Also, a manually-defined `schema.Schema` can be built rather than being loaded.

```
schema= stingray.schema.Schema(
    stingray.schema.Attribute( name='Column #1' ),
    stingray.schema.Attribute( name='Key' ),
    stingray.schema.Attribute( name='Value' ),
    stingray.schema.Attribute( name='Etc.' ),
)
```

1.7.4 Model

```
http://yuml.me/diagram/scruffy;/class/
#schema-loader,
[Schema]<>-[Attribute],
[SchemaLoader]-builds->[Schema],
[SchemaLoader]^[HeadingRowSchemaLoader],
[SchemaLoader]^[ExternalSchemaLoader],
[ExternalSchemaLoader]-reads->[Workbook],
[HeadingRowSchemaLoader]-reads->[Sheet].
```



1.7.5 Overheads

We depend on `schema`, `cell` and `sheet`.

```
"""stingray.schema.loader -- Loads a Schema from a row of a Sheet or
from a separate Sheet.  This is extended to load COBOL schema
from DDE files.
"""

from stingray.schema import Schema, Attribute
import stingray.cell
import stingray.sheet
```

1.7.6 Schema Loader

class `schema.loader.`**`SchemaLoader`**

A Schema Loader has one mandatory contract. A subclass may add a second contract.

1. It must load the schema.
2. An embedded schema loader will also return the non-schema rows.

```
class SchemaLoader:
    """Locate schema information.  Subclasses handle
    all of the variations on schema representation.
    """
    def __init__( self, sheet ):
        """A simple :py:class:`Sheet` instance."""
        self.sheet= sheet
        self.row_iter= iter( self.sheet.rows() )
    def schema( self ):
        """Scan the sheet to get the schema.
        :return: a :py:class:`Schema` object."""
        return NotImplemented
    def rows( self ):
        """Iterate all (or remaining) rows."""
        return self.row_iter
```

1.7.7 Embedded Schema Loader

class `schema.loader.`**`HeadingRowSchemaLoader`**

In many cases, the schema is first-row column titles or something similar. As we noted above, `csv.DictReader` supports this simple case.

All other cases have to be handled with something a bit more sophisticated. The `schema.loader.SchemaLoader` can be further subclassed to provide for more complex schema definitions buried in the rows of a sheet.

This means that we must make the schema parsing an application-provided plug-in that the Workbook uses when instantiating each Sheet.

```
class HeadingRowSchemaLoader( SchemaLoader ):
    """Read just the first row of a sheet to get embedded
    schema information."""
    def schema( self ):
        """Get the schema from row one.  Remaining rows are data."""
        row_1= next( self.row_iter )
        attributes = (
            dict(name=c.to_str()) for c in row_1
        )
        schema = Schema(
            *(Attribute(**col) for col in attributes)
        )
        return schema
```

We'll open a `sheet.Sheet` with a specific loader.

```
sheet= stingray.sheet.EmbeddedSchemaSheet(
    self.wb, 'The_Name',
    loader_class=HeadingRowSchemaLoader )
```

In many cases, we'd like to subclass this to suppress the empty rows that are an inevitable feature of workbook sheets. This doesn't work well for COBOL or Fixed format files, since an "empty" row may be difficult to discern.

```
class NonBlankHeadingRowSchemaLoader( HeadingRowSchemaLoader ):
    def __init__( self, sheet ):
        """A simple :py:class:`Sheet` instance."""
        self.sheet= sheet
        self.row_iter= self.non_blank( self.sheet.rows() )
    def non_blank( self, rows ):
        for r in rows:
            if all( c.is_empty() for c in r ):
                continue
            yield r
```

1.7.8 External Schema Loader

class `schema.loader.`**`ExternalSchemaLoader`**

In some cases, the data workbook is described by a separate schema workbook, or a separate sheet within the data workbook. In these cases, the other sheet (or file) must be parsed to locate schema information.

In the case of a fixed format file, we must examine a separate file to load schema information. This additional schems file may be in COBOL notation, leading to a more complex parser. See *COBOL Loader Module – Parse COBOL Source to Load a Schema*.

The layout of the schema, of course, will be highly variable, so the "meta-schema" must be adjusted to the actual file.

Note, also, that the schema loader is – itself – a typical of schema-based reader. It has a number of common features.

1. A dictionary-based "builder", `schema.loader.ExternalSchemaLoader.build_attr()`, to handle Logical Layout. This transforms the input "raw" dictionary of `cell.Cell` instances to an application dictionary of proper Python objects. See *The Stingray Developer's Guide*.
2. An iterator, `schema.loader.ExternalSchemaLoader.attr_dict_iter()`, that provides "raw" dictionaries from each row (based on the schema) to the builder to create application dictionaries.
3. The overall function, `schema.loader.ExternalSchemaLoader.schema()`, that iterates over application objects built from application dictionaries.

```
class ExternalSchemaLoader( SchemaLoader ):
    """Open a workbook file in a well-known format.
    Build a schema with attribute name, offset, size  and type
    information.  The type is a string that names the
    type of cell to create.

    The meta-schema must be embedded as the first line of the schema sheet.

    The assumed meta-schema is the following::

        Schema(
            Attribute("name",create="TextCell"),
            Attribute("offset",create="NumberCell"),
            Attribute("size",create="NumberCell"),
            Attribute("type",create="TextCell"),
        )

    If the meta-schema has different names, then a subclass with
    a different :py:meth:`build_attr` is required to map the actual
    source columns to the attributes of a :py:class:`Attribute`.
    Offsets are typically 1-based.
    """
    def __init__( self, workbook, sheet_name=’Sheet1’ ):
        self.workbook, self.sheet_name = workbook, sheet_name
        self.sheet= self.workbook.sheet( self.sheet_name, stingray.sheet.EmbeddedSchemaSheet,
        loader_class= HeadingRowSchemaLoader )
```

There’s potential for a great deal of variability in schema definition. Consequently, this `build_attr` method is merely a sample that covers one common case.

`ExternalSchemaLoader.`**`build_attr`**`(`*row*`)`

```
base= 1
type_to_cell = {
    ’text’: "TextCell",
    ’number’: "NumberCell",
    ’date’: "DateCell",
    ’boolean’: "BooleanCell",
    }
@staticmethod
def build_attr( row ):
    """Build application dictionary from raw dictionary.
    """
    try:
        offset= row[’offset’].to_int()-ExternalSchemaLoader.base
    except KeyError:
        offset= None
    try:
        size= row[’size’].to_int()
    except KeyError:
        size= None
    try:
        type_name= row[’type’].to_str()
        create= ExternalSchemaLoader.type_to_cell[type_name]
    except KeyError:
        create= stingray.cell.TextCell
    return dict(
        name= row[’name’].to_str(),
        offset= offset,
        size= size,
        create= create,
    )
```

Schema loading involves a process of

1. Iterating through the source rows as dictionaries.
   - Build each raw row as a source dictionary.
   - Build an standardized attr dictionary from the source dictionary. This mapping, implemented by `schema.loader.ExternalSchemaLoader.build_attr()` is subject to a great deal of change without notice.
2. Building each `schema.Attribute` from the dictionary.

`ExternalSchemaLoader.`**`attr_dict_iter`**`(`*sheet*`)`

```
def attr_dict_iter( self, sheet ):
    """Iterate over application dicts based on raw dicts
    built by the schema of the sheet."""
    return (
        ExternalSchemaLoader.build_attr(r)
        for r in sheet.schema.rows_as_dict_iter(sheet)
    )
```

ExternalSchemaLoader.**schema**()

```
def schema( self ):
    """Scan a file to get the schema.
    :return: a :py:class:`Schema` object."""
    self.row_iter= iter( [] )
    source_dict = self.attr_dict_iter( self.sheet )
    schema= Schema(
        *(Attribute(**row) for row in source_dict)
    )
    return schema
```

1.7.9 Worst-Case Loader

class schema.loader.**BareExternalSchemaLoader**

This is a degenerate case loader where the schema sheet (or file) doesn't have an embedded schema on line one of the sheet.

```
class BareExternalSchemaLoader( SchemaLoader ):
    """Open a workbook file in a well-known format.  Apply a schema parser
    to the given sheet (or file) to build a schema.

    The meta-schema is hard-coded in this class because the given
    sheet has no headers.
    """
    schema= Schema(
            Attribute("name",create="TextCell"),
            Attribute("offset",create="NumberCell"),
            Attribute("size",create="NumberCell"),
            Attribute("type",create="TextCell"),
        )
    def __init__( self, workbook, sheet_name='Sheet1' ):
        self.workbook, self.sheet_name = workbook, sheet_name
        self.sheet= self.workbook.sheet( self.sheet_name, stingray.sheet.ExternalSchemaSheet,
        schema= self.schema )
```

1.7.10 Parsing and Loading a COBOL Schema

One logical extension to this is to parse COBOL DDE's to create a schema that allows us to process a COBOL file (in EBCDIC) directly as if it were a simple workbook.

We'll delegate that to *COBOL Loader Module – Parse COBOL Source to Load a Schema*, since it's considerably more complex than simply loading rows from a sheet of a workbook.

1.8 Workbook Package – Uniform Wrappers for Workbooks

This package includes a base definition module plus a module for each physical format.

1.8.1 Workbook __init__ Module – Wrapper for all implementations

A *Workbook* is a collection of *Sheets*. It's also a set of decoding rules required to translate bytes (or XML text) into meaningful *Cell* instances.

Access to cells of a Workbook requires two levels of schema:

- *Physical Format*. The format required to locate cells. CSV, XLS, XLSX, ODS, are all well-known physical formats and the physical schema is implied by the file type. Fixed format and COBOL format, are not well-known, and a physical schema is required.
- *Logical Layout*. The columns or data elements present in the file. This may depend on an embedded schema in the first rows of a Sheet. Or it may depend on an external schema defined in another Workbook.

This package addresses the physical format issues. It provides a common abstraction over a number of forms of workbook data. It makes the physical format largely transparent to an application.

It's difficult to make the logical layout transparent. See *The Stingray Developer's Guide* for guidelines on developing applications that are flexible with respect to logical layout.

In a way, a Workbook is a factory for `sheet.Sheet` and `sheet.Row` objects.

More interestingly, a Workbook is a factory for `cell.Cell` instances. This is because the decoding of bytes to create a cell is entirely a feature of the Workbook.

Use Case

See *Introduction* for our physical-format independence use case. A `workbook.open_workbook()` function allows a program to be independent of physical format.

```
def process_workbook( input ):
    with workbook.open_workbook( input ) as source:
        process_workbook( source );

if __name__ == ‘‘__main__’’:
    application startup
    for input in args.file:
        process_workbook( input )
```

This does not address logical layout issues, however, which are handled by a `schema.Schema`. We might load an embedded schema or an external schema.

```
def process_sheet( sheet ):
    ‘‘’’‘‘Separated to facilitate unit testing’’‘‘’’
    counts= defaultdict( int )
    for rows in sheet.rows():
        process row
    return counts

def process_workbook( source ):
    for name in source.sheets():
        sheet= source.sheet( name,
            sheet.EmbeddedSchemaSheet,
            loader_class=schema.loader.HeadingRowSchemaLoader )
        counts= process_sheet( sheet )
        pprint.pprint( counts )
```

Physical Formats

Much data is transferred via formats tied to desktop spreadsheet software or informed by legacy mainframe design patterns. Data that comes from spreadsheet applications will have all the rich variety of desktop tools.

- CSV. This includes the "quote-comma" dialetcs as used by spreadsheets as well as "tab" or "pipe" dialetcs favored by Linux applications.
- ODS. This is a zipped archive of XML documents from which data can be extracted. This is an ECMA standard. This is the Open Office Spreadsheet structure. Most of the relevant data is in a content.xml member.
- XLSX or XLSM. This is a zipped archive of XML documents from which data can be extracted. This is an ECMA standard.
- XLS. This is the proprietary "Horrible Spreadsheet Format" (HSSF) as used by Microsoft products. We require xlrd to extract data from these files.
- Apple iWorks '09 Numbers formats. The iWorks '09 physical format is a simple ZipFile with a big XML document.
- Apple iWorks '13 Numbers formats. iWorks '13 physical format is the "bundle" or "package" format; the document is a directory, which contains a zip archive of .IWA files. These use snappy compression and protobuf object representation.
- Fixed Format, COBOL-style. Yes, these files still exist. For these files, schema information is *required* to determine where the fields are, since there's no puctuation. We can convert EBCDIC bytes or work in Unicode-compatible text. ASCII encoding is usually handled trivially by Python's `io` module.
- Other XML. For example, an Omni Outliner outlines with a normalized format. This is a possible future direction.

We'll call `CSV`, `XLS`, `XLSX` / `XLSM` and `ODS` the "well-known physical formats." They don't require physical schema information in order to identify the data items.

The Fixed and COBOL format files, on the other hand, require physical schema information. We'll look at COBOL in depth, in *The COBOL Package*.

iWork Numbers

The Stingray model of sheet/row/cell structure does not easily fit the Numbers sheet/table/row/cell structure.

Option 1:

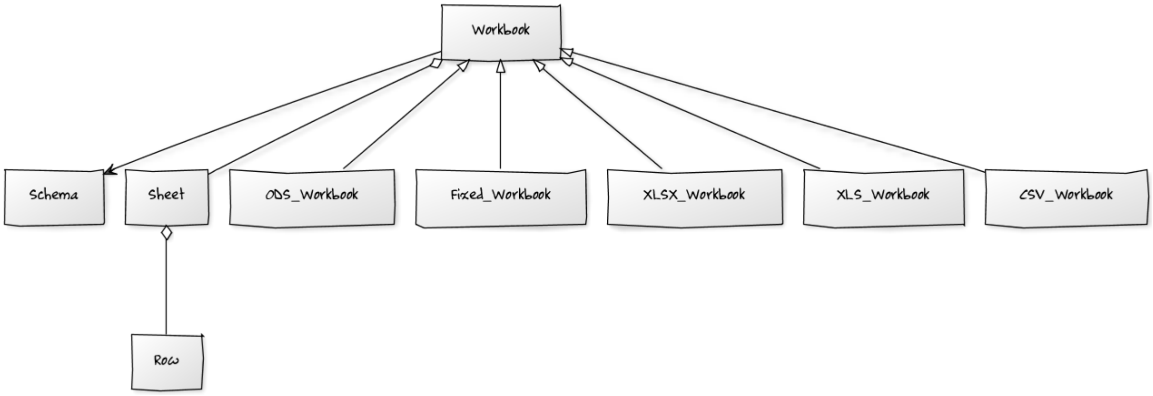
> Workbook -> new layer (Numbers "Workspace") -> Sheet (Numbers "Table") -> Row -> Cell

Option 2:

> Combine (Workspace,Table) into a 2-tuple, and call this a "sheet" name
>
> This will fit with Stingray acceptably.

Model

```
http://yuml.me/diagram/scruffy;/class/
#workbook,
[Workbook]^[CSV_Workbook],
[Workbook]^[XLS_Workbook],
[Workbook]^[XLSX_Workbook],
[Workbook]^[Fixed_Workbook],
[Workbook]^[ODS_Workbook],
[Workbook]<>-[Sheet],
[Sheet]<>-[Row],
[Workbook]->[Schema].
```



Implementation Overheads

Todo

Refactor workbook package

This module needs to be rebuilt into a package which imports a number of subsidiary modules. It's too large as written.

Adding Numbers '13 will make this module even more monstrous. Adding future spreadsheets will on exacerbate the problem.

It should become (like `cobol`) a high-level package that imports top-level classes from modules within the package.

> `from workbook.csv import CSV_Workbook`
>
> `from workbook.xls import XLS_Workbook`
>
> ... *etc.* ...

This should make a transparent change from module to package.

The top-level definition for `cobol.Workbook` must to be refactored into a `base` module that can be shared by all the modules in the package that extend this base definition.

A few Python overheads that must be physically first in the resulting module.

```
"""stingray.workbook -- Opens workbooks in various
formats, binds their associated schema, accesses them as Sheets with
Rows and Cells.

This is a kind of **Wrapper** or **Facade** that unifies :py:mod:`csv` and
:py:mod:`xlrd`. It handles a number of file formats including
:file:`.xlsx`, :file:`.ods`, and Numbers.
"""
```

We depend on the following

`xlrd` https://pypi.python.org/pypi/xlrd/0.9.2 http://www.lexicon.net/sjmachin/xlrd.htm

We'll rely on definitions of `cell`, `sheet`, and `schema.loader`. We have an implicit dependency on `schema`: we'll be given schema objects to work with.

```
import xlrd

import xml.etree.cElementTree as dom
from collections import defaultdict
import zipfile
import datetime
from io import open
import os.path
import pprint
import re
import glob
import logging
import decimal

import stingray.cell
import stingray.sheet
import stingray.schema.loader

from stingray.workbook.csv import CSV_Workbook
from stingray.workbook.xls import XLS_Workbook
from stingray.workbook.xlsx import XLSX_Workbook
from stingray.workbook.ods import ODS_Workbook
from stingray.workbook.numbers_09 import Numbers09_Workbook
from stingray.workbook.numbers_13 import Numbers13_Workbook
from stingray.workbook.fixed import Fixed_Workbook
```

Workbook Subclasses

We have a number of concrete subclasses of `workbook.Workbook`.

- `workbook.CSV_Workbook`. This is a degenerate case, where the workbook appears to contain a single sheet. This sheet is the CSV file, accessed via the built-in `csv.reader()`.
- `workbook.XLS_Workbook`. This is the workbook as processed by `xlrd`. These classes wrap `xlrd` classes to which the real work is delegated.
- `workbook.XLSX_Workbook`. This is the workbook after unzipping and using an XML parser on the various document parts. Mostly, this is a matter of unzipping and parsing parts of the document to create a DOM which can be traversed as needed.
- `workbook.Fixed_Workbook`. This is actually a fairly complex case. The workbook will appear to contain a single sheet; this sheet is the fixed format file. Schema information was required up front, unlike the other formats.
- `workbook.Numbers09_Workbook`. This handles the iWork '09 Numbers files with multiple workspaces and multiple tables in each workspace.
- `workbook.Numbers13_Workbook` These handle the iWork '13 Numbers files with multiple workspaces and multiple tables in each workspace.
- `workbook.ODS_Workbook`.

Extensions will handle various kinds of COBOL files. They're sumular to Fixed Workbooks.

We'd each of these to be a context manager, so we include the necessary methods.

Note that workbooks are rarely simple files. Sometimes they are ZIP archive members. Sometimes, they must be processed through gzip. Sometimes they involve Snappy compression.

In order to minimize the assumptions, we try to handle two forms of file processing:

- By name. In this case, the file name is provided. The file is opened and closed by the Workbook using the context manager interface.
- By file-like object. An open file-like object is provided. No additional context management is performed. This is appropriate when a workbook is itself a member of a larger archive.

Workbook Factory

class `workbook.`**`No_Schema`**

The `No_Schema` exception is raised if there's a problem loading a schema.

```
class No_Schema( Exception ):
    """A valid schema could not be loaded."""
    pass
```

class `workbook.`**`Opener`**

An opener **Factory** class. A subclass can extend this to handle other file extensions and physical formats.

```
class Opener:
    """An extensible opener that examines the file extension."""
    def __call__( self, name, file_object=None,
        schema_path='.', schema_sheet= None, **kw ):
        """Open a workbook.
        :param name: filename to open.
        :param file_object: File-like object to process.  If not
        provided the named file will be opened.
        :param schema_path: Directory with external schema files
        :param schema_sheet: A sheet in an external schema workbook.
        """
        _, ext = os.path.splitext( name )
        ext = ext.lower()
        if ext == ".xls": return XLS_Workbook( name, file_object )
        elif ext in ( ".xlsx", ".xlsm" ):
            return XLSX_Workbook( name, file_object )
        elif ext in ( ".csv", ):
            return CSV_Workbook( name, file_object, **kw )
        elif ext in ( ".tab", ):
            return CSV_Workbook( name, file_object, delimiter='\t', **kw )
        elif ext in ( ".ods", ):
            return ODS_Workbook( name, file_object )
        elif ext in ( ".numbers", ):
            # Directory? It's Numbers13_Workbook; Zipfile? It's Numbers09_Workbook
            if os.path.is_dir( name ):
                return Numbers13_Workbook( name, file_object )
            else:
                return Numbers09_Workbook( name, file_object )
        else:
            # Ideally somefile.schema is the file
            # and schema.csv can be tracked down.
            schema_pat= os.path.join(schema_path, ext[1:]+".*")
            schema_choices= glob.glob( schema_pat )
            if schema_choices:
                schema_name= schema_choices[0]
                schema_wb= open_workbook( schema_name )
                esl= stingray.schema.loader.ExternalSchemaLoader( schema_wb, schema_sheet )
                schema= esl.schema()
                return Fixed_Workbook( name, file_object, schema=schema )
            else:
                raise No_Schema( schema_pat )
```

workbook.**open_workbook** (*name*, *file_object*, *schema_path*, *schema_sheet*)

The default workbook.open_workbook() is simply an instance of the workbook.Opener.

```
open_workbook= Opener()
```

This allows a user to create subclasses to handle the various other file name extensions. Often, there are application-specific rules, or command-line options that will determine a mapping bewtween filename and physical format.

Also, an application may require external schema, or there may be an optional external schema with application-specific rules for handling this.

For fixed format files, we attempt to track down and load the relevant schema. An application might have narrower and more specific rules for binding file and schema. See below for the schema.loader.ExternalSchemaLoader class.

1.8.2 Workbook Base Definition

```
import logging
import os

import stingray.sheet
```

class workbook.**Workbook**

We note that these physical formats all encode a single, common data structure. Here are some abstract definitions.

```
class Workbook:
    """A workbook file; a collection of Sheets."""
    def __init__( self, name, file_object=None ):
        """Prepare the workbook for reading.

        :param name: File name
        :param file_object: Optional file-like object.  If omitted, the named file is opened.
        """
        self.name, self.file_obj= name, file_object
        self.the_file = None # Any internal files
        self.datemode= 0 # For xlrd
        self.log= logging.getLogger( self.__class__.__qualname__ )
    def __repr__( self ):
        return "{0}({1!r})".format( self.__class__.__qualname__, self.name )
```

Workbook.**sheet** (*sheet_name*, *sheet_type*, **args*, ***kw*)

There are two varieties of sheets, depending on the presence or absence of a schema.

```
def sheet( self, sheet_name, sheet_type=None, *args, **kw ):
    """Returns a :py:class:`sheet.Sheet`, ready for processing."""
    if sheet_type is None: sheet_type= stingray.sheet.Sheet
    sheet = sheet_type( self, sheet_name, *args, **kw )
    return sheet
```

Workbook.**sheets** ()

```
def sheets( self ):
    """List of sheet names.
    The filename is a handy default for CSV and Fixed files.
    """
    nm, _ = os.path.splitext( os.path.basename(self.name) )
    return [ nm ]
```

The Context Manager interface.

```
def __enter__( self ):
    return self
def __exit__( self, exc_type, exc_val, exc_tb ):
    if self.the_file:
        self.the_file.close()
    if exc_type is not None: return False
```

Workbook.**rows_of**(*sheet*)

```
def rows_of( self, sheet ):
    """An iterator over all rows of the given sheet."""
    raise NotImplementedError
```

Workbook.**row_get**(*row*, *attribute*)

```
def row_get( self, row, attribute ):
    """Create a Cell from the row's data."""
    raise NotImplementedError
```

There are distinct subclasses of `workbook.Workbook`, based on the physical file format.

Many of our physical formats don't require any physical schema information. A **Fixed** file, however, requires a physical format schema definition in order to decompose each line into cells.

1.8.3 CSV Workbook

```
import csv
import logging
import pprint

from stingray.workbook.base import Workbook
import stingray.sheet
import stingray.cell
```

class workbook.**CSV_Workbook**

We're wrapping the `csv.reader()`. We need to create proper `cell.TextCell` instances instead of the default string values that `csv` normally creates.

```
class CSV_Workbook( Workbook ):
    """Uses ``csv.reader``.  There's one sheet only."""
    def __init__( self, name, file_object=None, **kw ):
        """Prepare the workbook for reading.
        :param name: File name
        :param file_object: Optional file-like object.  If omitted, the named file is opened.
            If provided, it must be opened with  newline='' to permit non-standard
            line-endings.

        The kw are passed to :py:func:`csv.reader`
        to provide dialect information."""
        super().__init__( name, file_object )
        if self.file_obj:
            self.the_file= None
            self.rdr= csv.reader( self.file_obj, **kw )
        else:
            self.the_file = open( name, 'r', newline='' )
            self.rdr= csv.reader( self.the_file, **kw )
```

We can build an eager `sheet.Row` or a `sheet.LazyRow` from the available data. The eager Row includes the conversions. The `sheet.LazyRow` defers the conversions until the callback to `workbook.Workbook.row_get()`.

`CSV_Workbook.`**`rows_of`**(*sheet*)

```
def rows_of( self, sheet ):
    """An iterator over all rows of the named sheet.
    For CSV files, the sheet.name is simply ignored.
    """
    for data in self.rdr:
        logging.debug( pprint.pformat( data, indent=4 ) )
        row = stingray.sheet.Row( sheet, *(stingray.cell.TextCell(col,self) for col in data) )
        yield row
```

`CSV_Workbook.`**`row_get`**(*row*, *attribute*)

```
def row_get( self, row, attribute ):
    """Create a Cell from the row's data."""
    return row[attribute.position]
```

Since `csv` is eager, returning an individual `cell.TextCell` is easy.

1.8.4 XLS Workbook

```
import xlrd
import logging
import pprint

from stingray.workbook.base import Workbook
import stingray.sheet
import stingray.cell
```

class `workbook.`**`XLS_Workbook`**

This definition of a workbook wraps `xlrd` so that it fits the Stingray framework. We'll use proper `cell.Cell` subclass instances instead of the default `xlrd.Cell` values that `xlrd` normally creates.

```
class XLS_Workbook( Workbook ):
    """Uses ``xlrd``."""
    def __init__( self, name, file_object=None, **kw ):
        """Prepare the workbook for reading.
        :param name: File name
        :param file_object: Optional file-like object.  If omitted, the named file is opened.

        The kw arguments are passed to :py:func:`xlrd.open_workbook`.
        """
        super().__init__( name, file_object )
        if self.file_obj:
            self.wb= xlrd.open_workbook( self.name, file_contents=self.file_obj.read(), **kw )
        else:
            self.wb= xlrd.open_workbook( self.name, **kw )
        self.datemode= self.wb.datemode
```

XLS_Workbook.**sheets**()

```
def sheets( self ):
    """List of sheet names."""
    return self.wb.sheet_names()
```

We can build an eager `sheet.Row` or a `sheet.LazyRow` from the available data. The eager Row includes the conversions. The LazyRow defers the conversions until the callback to `XLS_Workbook.row_get()`.

XLS_Workbook.**rows_of**(*sheet*)

```
def rows_of( self, sheet ):
    """An iterator over all rows of the given sheet."""
    self.sheet= self.wb.sheet_by_name(sheet.name)
    for n in range(self.sheet.nrows):
        data = self.sheet.row(n)
        row = stingray.sheet.Row( sheet, *(self.cell(col) for col in data) )
        yield row
```

XLS_Workbook.**row_get**(*row*, *attribute*)

```
def row_get( self, row, attribute ):
    """Create a Cell from the row's data."""
    return row[attribute.position]
```

In `XLS_Workbook.rows_of()` we built a row eagerly. That way, returning an individual Cell is easy.

Convert a single `xlrd.Cell` to a proper subclass of `cell.Cell`

```
def cell( self, xlrd_cell ):
    if xlrd_cell.ctype == xlrd.XL_CELL_EMPTY:
        return stingray.cell.EmptyCell('', self)
    elif xlrd_cell.ctype == xlrd.XL_CELL_TEXT:
        return stingray.cell.TextCell( xlrd_cell.value, self )
    elif xlrd_cell.ctype == xlrd.XL_CELL_NUMBER:
        return stingray.cell.NumberCell( xlrd_cell.value, self )
    elif xlrd_cell.ctype == xlrd.XL_CELL_DATE:
        return stingray.cell.FloatDateCell( xlrd_cell.value, self )
    elif xlrd_cell.ctype == xlrd.XL_CELL_BOOLEAN:
        return stingray.cell.BooleanCell( xlrd_cell.value, self )
    elif xlrd_cell.ctype == xlrd.XL_CELL_ERROR:
        return stingray.cell.ErrorCell(
            xlrd.error_text_from_code[xlrd_cell.value], self )
    elif xlrd_cell.ctype == xlrd.XL_CELL_BLANK:
        return stingray.cell.EmptyCell('', self)
    else:
        raise ValueError( "Damaged Workbook" )
```

1.8.5 XLSX or XLSM Workbook

```
import logging
import pprint
import xml.etree.cElementTree as dom
import re
import zipfile
from collections import defaultdict

from stingray.workbook.base import Workbook
```

```
import stingray.sheet
import stingray.cell
```

class `workbook.`**`XLSX_Workbook`**

We're opening a ZIP archive and parsing the various XML documents that we find therein.

The `ElementTree` incremental parser provides parse "events" for specific tags, allowing for lower-memory parsing of the sometimes large XML documents.

See http://effbot.org/zone/element-iterparse.htm

The class as a whole defines some handy constants like XML namespaces and a pattern for parsing Cell ID's to separate the letters from the numbers.

```
class XLSX_Workbook( Workbook ):
    """ECMA Standard XLSX or XLSM documents.
    Locate sheets and rows within a given sheet.

    See http://www.ecma-international.org/publications/standards/Ecma-376.htm
    """
    # Relevant subset of namespaces used
    XLSX_NS = {
    "main":"http://schemas.openxmlformats.org/spreadsheetml/2006/main",
    "r":"http://schemas.openxmlformats.org/officeDocument/2006/relationships",
    "rel":"http://schemas.openxmlformats.org/package/2006/relationships",
    }
    cell_id_pat = re.compile( r"(\D+)(\d+)" )
    def __init__( self, name, file_object=None ):
        """Prepare the workbook for reading.
        :param name: File name
        :param file_object: Optional file-like object.  If omitted, the named file is opened.
        """
        super().__init__( name, file_object )
        self.zip_archive= zipfile.ZipFile( file_object or name, "r" )
        self._prepare()
```

The are two preparation steps required for reading these files. First, the sheets must be located. This involves resolving internal rID numbers. Second, the shared strings need to be loaded into memory.

```
def _prepare( self ):
    self._locate_sheets()
    self._get_shared_strings()
```

Locate all sheets involves building a `name_to_id` mapping and and `id_to_member` mapping. This allows is to map the user-oriented name to an id and the id to the XLSX zipfile member.

```
def _locate_sheets( self ):
    """Locate the name to id mapping and the id to member mapping.
    """
    # 1a. Open "workbook.xml" member.
    workbook_zip= self.zip_archive.getinfo("xl/workbook.xml")
    workbook_doc= dom.parse( self.zip_archive.open(workbook_zip) )
    # 1b. Get a dict of sheet names and their rIdx values.
    key_attr_id= 'name'
    val_attr_id= dom.QName( self.XLSX_NS['r'], 'id' )
    self.name_to_id = dict(
        ( s.attrib[key_attr_id], s.attrib[val_attr_id] )
        for s in workbook_doc.findall("*/main:sheet", namespaces=self.XLSX_NS)
    )
    logging.debug( self.name_to_id )

    # 2a. Open the "_rels/workbook.xml.rels" member
    rels_zip= self.zip_archive.getinfo("xl/_rels/workbook.xml.rels")
    rels_doc= dom.parse( self.zip_archive.open(rels_zip) )
    # 2b. Get a dict of rIdx to Target member name
    logging.debug( dom.tostring( rels_doc.getroot() ) )
    key_attr_id= ’Id’
    val_attr_id= ’Target’
    self.id_to_member = dict(
        ( r.attrib[key_attr_id], r.attrib[val_attr_id] )
        for r in rels_doc.findall("rel:Relationship", namespaces=self.XLSX_NS)
    )
    logging.debug( self.id_to_member )
```

Get Shared Strings walks a fine line. Ideally, we’d like to parse the document and simply use `itertext` to gather all of the text within a given string instance (`<si>`) tag. **However.**

In practice, these documents can be so huge that they don’t fit in memory comfortably. We rely on incremental parsing via the `iterparse` function.

```
def _get_shared_strings( self ):
    """Build ``strings_dict`` with all shared strings.
    """
    self.strings_dict= defaultdict(str)
    count= 0
    text_tag= dom.QName( self.XLSX_NS[’main’], "t" )
    string_tag= dom.QName( self.XLSX_NS[’main’], "si" )
    # 1. Open the "xl/sharedStrings.xml" member
    sharedStrings_zip= self.zip_archive.getinfo("xl/sharedStrings.xml")
    for event, element in dom.iterparse(
        self.zip_archive.open( sharedStrings_zip ), events=(’end’,) ):
        logging.debug( event, element.tag )
        if element.tag == text_tag:
            self.strings_dict[ count ]+= element.text
        elif element.tag == string_tag:
            count += 1
        element.clear()
    logging.debug( self.strings_dict )
```

The shared strings may be too massive for in-memory incremental parsing. We can create a temporary extract file to handle this case. Here’s the kind of code we might use.

```
with tempfile.TemporaryFile( ) as temp:
    self.zip_archive.extract( sharedStrings_mbr, temp.filename )
    for event, element in dom.iterparse( temp.filename ):
        process event and element
```

`XLSX_Workbook.`**`sheets`**`()`

```
def sheets( self ):
    return self.name_to_id.keys()
```

Translate a col-row pair from `(letter, number)` to proper 0-based Python index of `(row, col)`.

```
@staticmethod
def make_row_col( col_row_pair ):
    col, row = col_row_pair
    cn = 0
    for char in col_row_pair[0]:
        cn = cn*26 + (ord(char)-ord("A")+1)
    return int(row), cn-1
```

We can build an eager `sheet.Row` or a `sheet.LazyRow` from the available data. The eager `sheet.Row` is built from `cell.Cell` objects. The `sheet.LazyRow` delegates the creation of `cell.Cell` objects to `Workbook.row_get()`.

This uses an incremental parser, also. There are four kinds of tags that have to be located.

- `<row>`*row*`</row>`, end event. Finish (and yield) the row of cells. Since XLSX is sparse, missing empty cells must be filled in.
- `<c t="`*type*`" r="`*id*`">`*cell*`</c>`.
  - Start event for `c`. Get the cell type and id. Empty the value accumulator.
  - End event for `c`. Save the accumulated value. This allows the cell to have mixed content model.
- `<v>`*value*`</v>`, end event. Use the `cell()` method to track down enough information to build the Cell instance.

`XLSX_Workbook.`**`rows_of`**`(`*sheet*`)`

```
def rows_of( self, sheet ):
    """Iterator over rows as a list of Cells for a named worksheet."""
    # 1. Map user name to member.
    rId = self.name_to_id[sheet.name]
    self.sheet_member_name = self.id_to_member[rId]
    # 2. Open member.
    sheet_zip= self.zip_archive.getinfo("xl/"+self.sheet_member_name)
    self.row= {}
    # 3. Assemble each row, allowing for missing cells.
    row_tag= dom.QName(self.XLSX_NS['main'], "row")
    cell_tag= dom.QName(self.XLSX_NS['main'], "c")
    value_tag= dom.QName(self.XLSX_NS['main'], "v")
    format_tag= dom.QName(self.XLSX_NS['main'], "f")

    for event, element in dom.iterparse(
        self.zip_archive.open(sheet_zip), events=('start','end') ):
        logging.debug( element.tag, repr(element.text) )
        if event=='end' and element.tag == row_tag:
            # End of row: fill in missing cells
            if self.row.keys():
                data= stingray.sheet.Row( sheet, *(
                    self.row.get(i, stingray.cell.EmptyCell('', self))
                    for i in range(max(self.row.keys())+1) ) )
                yield data
            else:
                yield stingray.sheet.Row( sheet )
            self.row= {}
            element.clear()
        elif event=='end' and element.tag == cell_tag:
            # End of cell: consolidate the final string
            self.row[self.row_col[1]] = self.value
            self.value= stingray.cell.EmptyCell( '', self )
        elif event=='start' and element.tag == cell_tag:
            # Start of cell: collect a string in pieces.
            self.cell_type= element.attrib.get('t',None)
            self.cell_id = element.attrib['r']
            id_match = self.cell_id_pat.match( self.cell_id )
            self.row_col = self.make_row_col( id_match.groups() )
            self.value= stingray.cell.EmptyCell( '', self )
        elif event=='end' and element.tag == value_tag:
            # End of a value; what type was it?
            self.value= self.cell( element )

        elif event=='end' and element.tag == format_tag:
            pass # A format string
        else:
            pass
            logging.debug( "Ignoring", end="" ) # Numerous bits of structure exposed.
            logging.debug( dom.tostring(element) )
```

XLSX_Workbook.**row_get**(*row*, *attribute*)

```
def row_get( self, row, attribute ):
    """Create a Cell from the row's data."""
    return row[attribute.position]
```

Build a subclass of `cell.Cell` from the current value tag content plus the containing cell type information.

```
def cell( self, element ):
    """Create a proper :py:class:`cell.Cell` subclass from cell and value information."""
    logging.debug( self.cell_type, self.cell_id, element.text )
    if self.cell_type is None or self.cell_type == 'n':
        try:
            return stingray.cell.NumberCell( float(element.text), self )
        except ValueError:
            print( self.cell_id, element.attrib, element.text )
            return None
    elif self.cell_type == "s":
        try:
            # Shared String?
            return stingray.cell.TextCell( self.strings_dict[int(element.text)], self )
        except ValueError:
            # Inline String?
            logging.debug( self.cell_id, element.attrib, element.text )
            return stingray.cell.TextCell( element.text, self )
        except KeyError:
            # Not a valid shared string identifier?
            logging.debug( self.cell_id, element.attrib, element.text )
            return stingray.cell.TextCell( element.text, self )
    elif self.cell_type == "b":
        return stingray.cell.BooleanCell( float(element.text), self )
    elif self.cell_type == "d":
        return stingray.cell.FloatDateCell( float(element.text), self )
    elif self.cell_type == "e":
        return stingray.cell.ErrorCell( element.text, self )
    else:
        # 'str' (formula), 'inlineStr' (string), 'e' (error)
        print( self.cell_type, self.cell_id, element.attrib, element.text )
        logging.debug( self.strings_dict.get(int(element.text)) )
        return None
```

1.8.6 ODS Workbook

```
import logging
import pprint
import xml.etree.cElementTree as dom
import zipfile
import datetime

from stingray.workbook.base import Workbook
import stingray.sheet
import stingray.cell
```

class workbook.**ODS_Workbook**

We should use `iterparse` rather than simply parsing the entire document. If the document is large, then we can't hold it all in memory.

```
class ODS_Workbook( Workbook ):
    """Standard OOO ODS document.
    Locate sheets and rows within a given sheet.
    """
    ODS_NS = {
    "office":"urn:oasis:names:tc:opendocument:xmlns:office:1.0",
    "table":"urn:oasis:names:tc:opendocument:xmlns:table:1.0",
    "text":"urn:oasis:names:tc:opendocument:xmlns:text:1.0",
    }
    date_format = "%Y-%m-%d"
    def __init__( self, name, file_object=None ):
        """Prepare the workbook for reading.
        :param name: File name
        :param file_object: Optional file-like object.  If omitted, the named file is opened.
        """
        super().__init__( name, file_object )
        self.zip_archive= zipfile.ZipFile( file_object or name, "r" )
        self._prepare()
```

As preparation for reading these files, we locate all the sheet names.

```
def _prepare( self ):
    self._locate_sheets()
```

Locating all the sheets is a matter of doing an XPath search for `body/spreadsheet/table` and getting the *name* attribute from the `<table name="`*name*`">` tags.

```
def _locate_sheets( self ):
    """Create ``tables`` map from name to table."""
    workbook_zip= self.zip_archive.getinfo("content.xml")
    workbook_doc= dom.parse( self.zip_archive.open(workbook_zip) )
    name_attr_id= dom.QName( self.ODS_NS["table"], "name" )
    logging.debug( dom.tostring( workbook_doc.getroot() ) )
    self.tables= dict(
        (t.attrib[name_attr_id],t)
        for t in workbook_doc.findall("office:body/office:spreadsheet/table:table",
            namespaces=self.ODS_NS) )
```

An `iterparse` version to locate sheets would look for start of `table` tags and then get the name attribute from that tag.

ODS_Workbook.**sheets**()

```
def sheets( self ):
    return self.tables.keys()
```

We can build an eager `sheet.Row` or a `sheet.LazyRow` from the available data. The eager Row includes the conversions. The LazyRow defers the conversions to `ODS_Workbook.row_get()`.

In ODS documents, the cell’s value can be carried in the value attribute or it can be a mixed content value of the element. There are three cases.

- <table-cell value-type="*type*" value="*value*">...</table-cell>
- <table-cell value-type="*type*" date-value="*value*">...</table-cell>
- <table-cell value-type="*type*">*value*</table-cell>

ODS_Workbook.**rows_of**()

```
def rows_of( self, sheet ):
    """Iterator over rows as a list of Cells for a named worksheet."""
    for r, row_doc in enumerate(
        self.tables[sheet.name].findall( "table:table-row", namespaces=self.ODS_NS ) ):
        row= []
        for c, cell_doc in enumerate( row_doc.findall( "table:table-cell", namespaces=self.ODS_NS ) )
            row.append( self.cell(cell_doc) )
        yield row
```

ODS_Workbook.**row_get**(*row*, *attribute*)

```
def row_get( self, row, attribute ):
    """Create a Cell from the row’s data."""
    return row[attribute.position]
```

Build a subclass of `cell.Cell` from the current type name and value.

Todo

Refactor this, it feels clunky.

```
def cell( self, cell_doc ):
    logging.debug( dom.tostring(cell_doc) )
    value_attr_id= dom.QName( self.ODS_NS[’office’], ’value’ )
    date_attr_id= dom.QName( self.ODS_NS[’office’], ’date-value’ )
    type_attr_id= dom.QName( self.ODS_NS[’office’], ’value-type’ )
    # Get the type
    try:
        type_name= cell_doc.attrib[type_attr_id]
    except KeyError:
        return stingray.cell.EmptyCell( ’’, self )
    value= None
    # Date value as attribute?
    if not value:
        try:
            value= cell_doc.attrib[date_attr_id]
        except KeyError:
            pass
    # Other value as attribute?
    if not value:
        try:
            value= cell_doc.attrib[value_attr_id]
        except KeyError:
            pass
    # No value attributes, get *all* the text content.
    if not value:
        value= "".join( x for x in cell_doc.itertext() )
    if not value:
        # TODO: Proper warning.
        dom.dump( cell_doc )
    logging.debug( type_name, repr(value) )
    if type_name == "string":
        return stingray.cell.TextCell( value, self )
    elif type_name == "float":
        return stingray.cell.NumberCell( float(value), self )
    elif type_name == "date":
        theDate= datetime.datetime.strptime(
            value, ODS_Workbook.date_format )
        return stingray.cell.FloatDateCell( theDate, self )
    elif type_name == "boolean":
        return stingray.cell.BooleanCell(
            float(value.upper()=='TRUE'),  self )
    elif type_name == "empty":
        return stingray.cell.EmptyCell( '', self )
    else:
        raise Exception( "Unknown cell {0}".format( dom.tostring(cell_doc) ) )
```

An `iterparse` version of building a row would look for start of `table` tags and then get the name attribute from that tag just to locate the right sheet.

Once the sheet was located, then the row and cell tags would be used

- At `<table-row` start: increment row number, reset buffer
- At `<table-row` end: yield the row
- At `<table-cell` start: check for empty, date, float, boolean types, which are available as an attribute at start. For strings, start accumulating string values.
- At `<table-cell` end: finalize the accumulated value.

1.8.7 Numbers 09 Workbook

```
import logging
import pprint
import xml.etree.cElementTree as dom
import zipfile
import datetime
import decimal

from stingray.workbook.base import Workbook
import stingray.sheet
import stingray.cell
```

The iWork 09 format is a (simpler) Zip file with an XML document inside it. There may be slight variations between native Numbers '09 and Numbers '13 doing a "save as" in Numbers '09 format.

Numbers '13 is entirely different. See *Numbers 13 Workbook*.

class `workbook.`**`Numbers09_Workbook`**

```
class Numbers09_Workbook( Workbook ):
    """Mac OS X Numbers Workbook for iWork 09.

    The ``.numbers`` "file" is a ZIP file.
    The index.xml is the interesting part of this.
    """
    NUMBERS_NS = {
    "ls":"http://developer.apple.com/namespaces/ls",
    "sf":"http://developer.apple.com/namespaces/sf",
    "sfa":"http://developer.apple.com/namespaces/sfa",
    }
    row_debug= False
    def __init__( self, name, file_object=None ):
        """Prepare the workbook for reading.
        :param name: File name
        :param file_object: Optional file-like object. Ignored for v3.2 numbers files.
        """
        super().__init__( name, file_object )
        self.zip_archive= zipfile.ZipFile( file_object or name, "r" )
        self._prepare()
```

As preparation for reading these files, we locate all the sheet names and all the number styles.

```
def _prepare( self ):
    """Locate sheets/tables and styles."""
    root= dom.parse( self.zip_archive.open('index.xml') ).getroot()
    self._locate_sheets(root)
    self._get_styles(root)
```

Locating all the sheets is a matter of doing an XPath search for `workspace-array/workspace` and getting the `workspace-name` attribute from the `<table name="`*name*`">` tags.

Within each workspace we have to find `page-info/tabular-info/tabular-model` to get the tables within the workspaces.

```
def _locate_sheets( self, root ):
    """Create ``workspace_table`` map from name to workspace and table."""
    self.workspace= dict()

    ws_name_attr= dom.QName( self.NUMBERS_NS["ls"], 'workspace-name' )
    name_attr= dom.QName( self.NUMBERS_NS["sf"], 'name' )
    workspace_array= root.find("ls:workspace-array", namespaces=self.NUMBERS_NS )
    for workspace in workspace_array.findall('.//ls:workspace', namespaces=self.NUMBERS_NS ):
        # Populate tables within this workspace.
        tables= dict()
        page_info = workspace.find('ls:page-info', namespaces=self.NUMBERS_NS)
        for tabular_info in page_info.findall('.//sf:tabular-info', namespaces=self.NUMBERS_NS):
            tabular_model = tabular_info.find( 'sf:tabular-model', namespaces=self.NUMBERS_NS)
            tables[ tabular_model.get(name_attr) ] = tabular_model
        self.workspace[ workspace.get(ws_name_attr) ]= workspace, tables
```

Locate a "data source" within the XML document. Create `Cell` instances.

```
def _datasource( self, grid ):
    """The data source for cell values within a grid.
    This yields each individual cell value, transformed into
    string, Decimal, datetime.
    """
    datasource = grid.find('.//sf:datasource', namespaces=self.NUMBERS_NS)
    for cell_doc in datasource:
        yield self.cell( cell_doc )
    # or return map( self.cell, datasource )
```

Create a `Cell` instance from the decoded data.

```
def cell( self, cell ):
    logging.debug( dom.tostring(cell) )

    date_tag= dom.QName( self.NUMBERS_NS["sf"], ’d’ )
    date_attr= dom.QName( self.NUMBERS_NS["sf"], ’cell-date’ )
    formula_tag= dom.QName( self.NUMBERS_NS["sf"], ’f’ )
    s_attr= dom.QName( self.NUMBERS_NS["sf"], ’s’ )
    v_attr= dom.QName( self.NUMBERS_NS["sf"], ’v’ )
    general_tag= dom.QName( self.NUMBERS_NS["sf"], ’g’ )
    number_tag= dom.QName( self.NUMBERS_NS["sf"], ’n’ )
    text_tag= dom.QName( self.NUMBERS_NS["sf"], ’t’ )
    o_tag= dom.QName( self.NUMBERS_NS["sf"], ’o’ )
    span_tag= dom.QName( self.NUMBERS_NS["sf"], ’s’ )
    bool_tag= dom.QName( self.NUMBERS_NS["sf"], ’b’ )
    popup_menu_tag= dom.QName( self.NUMBERS_NS["sf"], ’pm’ )
    IDREF_attr= dom.QName( self.NUMBERS_NS["sfa"], ’IDREF’ )
    ID_attr= dom.QName( self.NUMBERS_NS["sfa"], ’ID’ )
    fs_attr= dom.QName( self.NUMBERS_NS["sf"],"fs")

    if cell.tag == date_tag:
        seconds= int(cell.attrib[date_attr])
        epoch= datetime.datetime(2001, 1, 1)
        delta= datetime.timedelta( seconds=seconds )
        theDate= epoch + delta
        return stingray.cell.DateCell( theDate, self )

    elif cell.tag == formula_tag: # formula or error.
        s= cell.get(s_attr)
        fo= cell.find(’sf:fo’, namespaces=self.NUMBERS_NS)
        # Numeric Result? What about non-numeric results?
        r= cell.find(’sf:r’, namespaces=self.NUMBERS_NS)
        if r:
            # Result:
            rn= r.find(’sf:rn’, namespaces=self.NUMBERS_NS)
            try:
                value_txt= rn.attrib[v_attr]
                value= self._to_decimal( value_txt, s )
            except KeyError as ex:
                #self._cell_warning("Formula with no value", cell)
                value= self._to_decimal( ’0’, s )
            return stingray.cell.NumberCell( value, self )
        else:
            # Error:
            #self._cell_warning("Formula error", cell)
            value= "#Error in {0}".format(fo.get(fs_attr))
            return stingray.cell.ErrorCell( value, self )

    elif cell.tag == general_tag: # General?
        return stingray.cell.EmptyCell( ’’, self )
    elif cell.tag == number_tag: # number
        value= self._decode_number( cell )
        return stingray.cell.NumberCell( value, self )
    elif cell.tag == o_tag: #??
        self._cell_warning("Unknown cell type", cell)
        return stingray.cell.EmptyCell( ’’, self )
    elif cell.tag == span_tag: #span?
        self._cell_warning("Unknown cell type", cell)
        return stingray.cell.EmptyCell( ’’, self )
    elif cell.tag == text_tag: # text
        value= self._decode_text( cell )
        return stingray.cell.TextCell( value, self )
    elif cell.tag == bool_tag: # boolean
        value= self._decode_number( cell )
        return stingray.cell.BooleanCell( value, self )
    elif cell.tag == popup_menu_tag: # popup menu
        # TODO:: Better Xpath query: ‘‘menu-choices/*[@ID=’name’]‘‘
        value= None # In case we can’t find anything.
        selected= cell.find(’sf:proxied-cell-ref’, namespaces=self.NUMBERS_NS)
        name= selected.get(IDREF_attr)
        mc= cell.find(’sf:menu-choices’, namespaces=self.NUMBERS_NS)
        for t in mc:
            if t.get(ID_attr) == name:
                # t’s tag cold end in Could be "t", or "n".
                if t.tag.endswith(’t’): # Text
                    value= self._decode_text( t )
                    return stingray.cell.TextCell( value, self )
                elif t.tag.endswith(’n’): # Number
                    value= self._decode_number( t )
                    return stingray.cell.NumberCell( value, self )
                else:
                    raise Exception( "Unknown popup menu {0}".format(dom.tostring(cell)))
    else:
        raise Exception( "Unknown cell {0}".format( dom.tostring(cell) ) )
```

Some lower-level conversions.

```
def _to_decimal( self, value_txt, style_id ):
    """Convert a given numeric value_text using the named style.

    TODO: From the style, get the number of decimal places, use that to
    build a string version of the float value.
    """
    fdp_attr= dom.QName( self.NUMBERS_NS["sf"], ’format-decimal-places’ )
    fs_attr= dom.QName( self.NUMBERS_NS["sf"], ’format-string’ )
    cell_style= self.cell_style.get(style_id)
    #print( "TO_DECIMAL", value_txt, style_id, "=", cell_style )

    fs= None # cell_style.get(fs_attr) # Doesn’t seem correct
    fdp= None # cell_style.get(fdp_attr) # Doesn’t seem correct
    # Transform fs into proper Python format, otherwise, use the number of
    # decimal places.
    if fs is not None:
        fmt= self._rewrite_fmt( fs )
        #print( "Decimal: {{0:{0}}}.format({1}) = ".format( fmt, value_txt ), end="" )
        value= decimal.Decimal( "{:{fmt}}".format(float(value_txt), fmt=fmt) )
        #print( value )
        return value
    elif fdp is not None:
        #fmt= "{{0:.{0}f}}".format(fdp)
        value= decimal.Decimal( "{:.{fdp}f}".format(float(value_txt), fdp=fdp) )
        #print( "Decimal: {0}.format({1}) = {2!r}".format( fmt, value_txt, value ) )
        return value
    else:
        value= decimal.Decimal( value_txt )
        #print( "Decimal: {0} = {1!r}".format( value_txt, value ) )
    return value
```

```
def _decode_text( self, cell ):
    """Decode a <t> tag's value."""
    sfa_s_attr= dom.QName( self.NUMBERS_NS["sfa"], 's' )
    ct= cell.find( 'sf:ct', namespaces=self.NUMBERS_NS )
    value= ct.get(sfa_s_attr)
    if value is None:
        value= "\n".join( cell.itertext() )
    return value

def _decode_number( self, cell ):
    """Decode a <n> tag's value, applying the style."""
    s_attr= dom.QName( self.NUMBERS_NS["sf"], 's' )
    v_attr= dom.QName( self.NUMBERS_NS["sf"], 'v' )
    s= cell.get(s_attr)
    cell_style= self.cell_style.get(s)
    try:
        value_txt= cell.attrib[v_attr]
        value= self._to_decimal( value_txt, s )
    except KeyError as ex:
        #self._cell_warning("Number with no value", cell)
        value= self._to_decimal( '0', s )
    return value
```

The styles are also important because we can use them to parse the numbers more precisely.

```
def _get_styles( self, root ):
    """Get the styles."""
    ID_attr= dom.QName( self.NUMBERS_NS["sfa"], 'ID' )
    ident_attr= dom.QName( self.NUMBERS_NS["sf"], 'ident' )
    parent_ident_attr= dom.QName( self.NUMBERS_NS["sf"], 'parent-ident' )

    self.cell_style= {}
    for cs in root.findall('.//sf:cell-style', namespaces=self.NUMBERS_NS):
        #print( "STYLE", dom.tostring(cs) )
        ID= cs.get(ID_attr)
        ident= cs.get(ident_attr)
        parent_ident= cs.get(parent_ident_attr)
        property_number_format= cs.find('.//sf:SFTCellStylePropertyNumberFormat', namespaces=self.NUM
        if property_number_format is None:
            if parent_ident is not None:
                self.cell_style[ID]= self.cell_style[parent_ident]
        else:
            number_format= property_number_format.find('sf:number-format', namespaces=self.NUMBERS_NS
            if number_format is None:
                if parent_ident is not None:
                    self.cell_style[ID]= self.cell_style[parent_ident]
            else:
                self.cell_style[ID]= number_format.attrib
                if ident is not None:
                    self.cell_style[ident]= number_format.attrib
            #print( ID, self.cell_style.get(ID,None) )
```

Rewrite a number format from Numbers to Python

```
def _rewrite_fmt( self, format_string ):
    """Parse the mini-language: '#,##0.###;-#,##0.###' is an example.
    This becomes "{:10,.3f}"
    """
    positive, _, negative = format_string.partition(";")
    fmt= negative or positive
    digits= len(fmt)
    comma= "," if "," in fmt else ""
    whole, _, frac= fmt.partition(".")
    precision= len(frac)
    return "{digits}{comma}.{precision}f".format(
        digits= digits, comma=comma, precision=precision )
```

The “sheets” are the `[ (` *workspace, table* `), ... ]` pairs.

```
def sheets( self ):
    """Build "sheet" names from workspace/table"""
    sheet_list= []
    for w_name in self.workspace:
        ws, tables = self.workspace[w_name]
        for t_name in tables:
            sheet_list.append( (w_name, t_name) )
    return sheet_list
```

Picking a sheet involves matching a two-part name: (workspace, table).

```
def rows_of( self, sheet ):
    """Iterator over rows.

    Two parallel traversals:

    Internal iterator over grid/datasource/* has d, t, n, pm, g, o and s
        yields individual cell values.

    Iterator over grid/rows/grid-row may have ``nc``, number of columns in that row.
        Each grid-row fetches a number of cell values to assemble a row.
        Row's may be variable length (sigh) but padded to the number of columns
        specified in the grid.

    :param sheet: a Sheet object to retrieve rows from.
    """
    self.log.debug( "rows of {0}: {1}".format(sheet, sheet.name) )
    ws_name, t_name = sheet.name
    ws, tables= self.workspace[ws_name]
    tabular_model= tables[t_name]

    grid= tabular_model.find( 'sf:grid', namespaces=self.NUMBERS_NS )
    numrows_attr= dom.QName( self.NUMBERS_NS["sf"], 'numrows' )
    numcols_attr= dom.QName( self.NUMBERS_NS["sf"], 'numcols' )
    numrows = int(grid.attrib[numrows_attr])
    numcols = int(grid.attrib[numcols_attr])

    nc_attr= dom.QName( self.NUMBERS_NS["sf"], 'nc' )

    datasource= iter( self._datasource(grid) )

    rows = grid.find('sf:rows', namespaces=self.NUMBERS_NS)
    for n, r in enumerate(rows.findall( 'sf:grid-row', namespaces=self.NUMBERS_NS )):
        #print( "ROW", dom.tostring(r) )
        self.debug_row= n
        # Is this really relevant for Numbers '09?
        nc= int(r.get(nc_attr,numcols))
        try:
            row= [ next(datasource) for self.debug_col in range(nc) ]
        except StopIteration as e:
            pass # Last row will exhaust the datasource.
        if len(row) == numcols:
            yield row
        else:
            yield row + (numcols-nc)*[None]
```

1.8.8 Numbers 13 Workbook

```
import logging
import pprint
import zipfile
import datetime
import decimal
import os
import struct

from stingray.workbook.base import Workbook
import stingray.sheet
import stingray.cell
from stingray.snappy import Snappy
from stingray.protobuf import Archive_Reader, Message
```

Todo

Additional Numbers13_Workbook Feature

Translate Formula and Formula error to Text

The iWork 13 format is a directory with an `index.zip` file. The ZIP contains a number of `.IWA` files. Each `.IWA` is compressed using the Snappy protocol. The uncompressed data is messages in Protobuf format.

We could depend on proper Snappy and Protobuf implementations. We provide our own fall-back implementation in case there's nothing better available.

We should import other implementations first, and then fall back to our own implementation. Instead, we'll simply import a local snappy and protobuf reader.

class `workbook.`**`Numbers13_Workbook`**

```
class Numbers13_Workbook( Workbook ):
    """Mac OS X Numbers Workbook for iWork '13.

    The ``.numbers`` "file" is a directory bundle or package.
    The ``index.zip`` file is the interesting part of the bundle.
    """
    def __init__( self, name, file_object=None ):
        """Prepare the workbook for reading.

        :param name: File name
        :param file_object: Ignored for iWork13 files.

        We might be able to use the file's internal handle to open
        it as a proper directory. But we don't.
        """
        super().__init__( name, None )
        self.archive= self.load_archive( name )
```

Read the archive to get the serialized messages. This method deserializes all of the protobuf-encoded messages. It’s a shabby stand-in for proper protobuf processing.

One thing we *could* do is to refactor this method into a bunch of methods, each of which is tied to a specific class of message. That would parallel the way protobuf really works.

```
@staticmethod
def load_archive( filename ):
    """Extract all the protobuf-serialized Archived messages.
    We don’t actually need to read **all** of them.
    We really only need Index/Document.iwa, Index/CalculationEngine.iwa, and
    all Index/Tables/*.iwa. But that’s almost everything.

    :param filename: File name
    """
    log= logging.getLogger( "load_archive" )
    snappy= Snappy()
    reader= Archive_Reader()
    archive=dict()
    with zipfile.ZipFile( os.path.join( filename, "index.zip" ) ) as index:
        for n in index.namelist():
            log.info( n )
            with index.open( n ) as member:
                data= snappy.decompress( member )
                log.debug( "{0} bytes".format(len(data)) )
                for id, m in reader.archive_iter( data ):
                    log.debug( "{0:4d}: {1}".format( id, m ) )
                    archive[id]= m
                    if m.name_ == "TN.DocumentArchive":
                        m.sheets= [Message("Reference", sheet) for sheet in m[1]]
                        log.debug( "{0} {1}".format(m.name_, m.sheets) )
                    elif m.name_ == "TN.SheetArchive":
                        m.name= bytes(m[1][0]).decode("UTF-8")
                        m.drawable_infos= [Message(’Reference’, ref) for ref in m[2]]
                        log.debug( "{0} {1} {2}".format(m.name_, m.name, m.drawable_infos) )
                    elif m.name_ == "TST.TableInfoArchive":
                        m.super= Message( ’DrawableArchive’, m[1][0] )
                        m.tableModel= Message(’Reference’, m[2][0])
                        log.debug( "{0} {1} {2}".format(m.name_, m.super, m.tableModel) )
                    elif m.name_ == "TST.TableModelArchive":
                        m.data_store= Message( "DataStore", m[4][0] )
                        m.table_id= bytes(m[1][0])
                        m.table_name= bytes(m[8][0]).decode("UTF-8")
                        log.debug( "{0} {1} {2}".format(m.name_, m.table_name, m.data_store) )
                        m.data_store.tiles= [Message("TileStorage", tile) for tile in m.data_store[3
                        log.debug( "{0} {1}".format(m.name_,m.data_store.tiles) )
                        for ts in m.data_store.tiles:
                            ts.tiles= [Message("TileStorage.Tile", tile) for tile in ts[1]]
                            for tile in ts.tiles:
                                tile.id= tile[1][0]
                                tile.ref= Message(’Reference’, tile[2][0])
                                log.debug( "{0} {1} {2} {3}".format(m.name_, tile, tile.id, tile.ref
                        m.data_store.stringTable= [Message(’Reference’, string) for string in m.data_
                        m.data_store.formulaTable= [Message(’Reference’, formula) for formula in m.da
                        m.data_store.formulaErrorTable= [Message(’Reference’, error) for error in m.
                        log.debug( "DataStore stringTable {0}, formulaTable {1}, formulaErrorTable {2
                            m.data_store.stringTable, m.data_store.formulaTable,
                            m.data_store.formulaErrorTable) )
                    elif m.name_ == "TST.TableDataList":
                        m.listType= m[1][0]
                        m.nextListID= m[2][0]
                        m.entries= [Message('ListEntry', entry) for entry in m[3]]
                        log.debug( "{0} {1} {2}".format( m.name_, m.listType, m.nextListID ) )
                        for entry in m.entries:
                            entry.key= entry[1][0]
                            try:
                                entry.string= bytes(entry[3][0])
                            except IndexError:
                                entry.string= b''
                            try:
                                entry.formula= Message('FormulaArchive', entry[5][0])
                                entry.formula.AST_node_array= Message("ASTNodeArrayArchive", entry.fo
                                entry.formula.AST_node_array.AST_node= [Message("AST_node", n) for n
                            except IndexError:
                                entry.formula= None
                            log.debug( "ListEntry {0} {1} {2}".format( entry, entry.string, entry.for
                    elif m.name_ == "TST.Tile":
                        m.rowInfos= [ Message("TileRowInfo", row_info) for row_info in m[5] ]
                        for row_info in m.rowInfos:
                            row_info.tileRowIndex= row_info[1][0]
                            row_info.cellCount= row_info[2][0]
                            row_info.cellStorageBuffer= row_info[3][0]
                            row_info.cellOffsets= row_info[4][0]
                            log.debug( "tileRowIndex {0} cellCount {1}".format(row_info.tileRowIndex,
                            format= "<{0}h".format(len(row_info.cellOffsets)//2)
                            offsets= struct.unpack( format, bytes(row_info.cellOffsets) )
                            log.debug( "{0} {1}".format( m.name_, offsets ) )
                            row_info.cells= dict()
                            count= row_info.cellCount
                            for col, offset in enumerate( offsets ):
                                if offset == -1:
                                    log.debug( "{0} {1} {2}".format( m.name_, col, offset ) )
                                    row_info.cells[col]= (None,None)
                                    continue
                                # A little needless copying to simplify the elif sequence.
                                cell_raw= row_info.cellStorageBuffer[offset:offset+32]
                                version, celltype = struct.unpack( "<hbx", bytes(cell_raw[:4]) )
                                if celltype == 2: # Number
                                    data= struct.unpack( "<IIIdi", bytes(cell_raw[4:28]) )
                                elif celltype == 3: # Text (in the associated TableDataList)
                                    data= struct.unpack( "<IIIi", bytes(cell_raw[4:20]) )
                                elif celltype == 5: # Date
                                    data= struct.unpack( "<IIIdi", bytes(cell_raw[4:28]) )
                                elif celltype == 6: # Boolean
                                    data= struct.unpack( "<IIId", bytes(cell_raw[4:24]) )
                                elif celltype == 8: # Error
                                    data= struct.unpack( "<hhII", bytes(cell_raw[4:16]) )
                                else:
                                    raise Exception( "Unsupported {0}".format(celltype) )
                                log.debug( "{0} {1} {2} {3} {4}".format( m.name_, col, offset, cellty
                                row_info.cells[col]= (celltype,data[3])
                                count -= 1
                                if count == 0: break

    return archive
```

Once we've decoded the archive, we can fetch sheets, tables, rows and cells.

We start with document.iwa and get the message with an id of 1. This is the TN.DocumentArchive and it lists the TN.SheetArchive by id.

Each TN.SheetArchive has drawable_infos references to TST.TableInfoArchive. TST.TableInfoArchive has reference to TST.TableModelArchive (and a super reference to SheetArchive.) TST.TableModelArchive references a DataStore.

TST.DataStore.stringTable references a TST.TableDataList. This has string literals or formulas (or styles) referenced by cells.

TST.DataStore.tiles references a TST.Tile. The Tile has some bytes which contain the cells as hard-to-parse raw structures. This includes text cells with references to the data lists string literals. It includes number, date and boolean cells with numeric-looking data.

```python
def sheets( self ):
    sheet_list= []
    document = self.archive[1]
    for sheet_ref in document.sheets:
        sheet= self.archive[sheet_ref[1][0]]
        for table_ref in sheet.drawable_infos:
            tableinfo= self.archive[table_ref[1][0]]
            tablemodel= self.archive[tableinfo.tableModel[1][0]]
            sheet_list.append(  (sheet.name, tablemodel.table_name) )
    return sheet_list
```

The proto files include this.

```
enum CellType {
  genericCellType = 0;
  spanCellType = 1;
  numberCellType = 2;
  textCellType = 3;
  formulaCellType = 4;
  dateCellType = 5;
  boolCellType = 6;
  durationCellType = 7;
  formulaErrorCellType = 8;
  automaticCellType = 9;
}
```

```python
def _cell( self, cell, strings, formulae, errors ):
    """Given a decoded row cell message, pluck out the relevant data.
    Look into the collection of strings or formulae for the data object.
    :param cell: cell message
    :param strings: dict of strings
    :param formulae: dict of formulae
    :param errors: dict of errors
    :returns: Cell object
    """
    celltype, value = cell
    if celltype == 0: # What does "generic" mean?
        v= stingray.cell.NumberCell( value, self )
    elif celltype == 1 or celltype is None:
        v= stingray.cell.EmptyCell( "", self )
    elif celltype == 2:
        v= stingray.cell.NumberCell( value, self )
    elif celltype in (3, 9): # Automatic??
        v= stingray.cell.TextCell( strings[value].decode("UTF-8"), self )
    elif celltype == 5:
        seconds= int(value)
        epoch= datetime.datetime(2001, 1, 1)
        delta= datetime.timedelta( seconds=seconds )
        theDate= epoch + delta
        v= stingray.cell.DateCell( theDate, self )
    elif celltype == 6:
        v= stingray.cell.BooleanCell( value, self )
    elif celltype == 7: # Actually a duration
        v= stingray.cell.NumberCell( value, self )
    elif celltype in (4, 8):
        ast_0= formulae[value][0]
        v= stingray.cell.ErrorCell( "Formula {0}".format(ast_0[1]), self )
    else:
        raise Exception( "Unknown cell type {0!r}".format(cell) )
    self.log.debug( "_cell {0} = {1}".format( cell, v) )
    return v

def rows_of( self, sheet ):
    """Iterator over rows.

    :param sheet: a Sheet object to retrieve rows from.
    """
    self.log.debug( "rows of {0}: {1}".format(sheet, sheet.name) )
    document = self.archive[1]
    for sheet_ref in document.sheets:
        sheet_msg= self.archive[sheet_ref[1][0]]
        for table_ref in sheet_msg.drawable_infos:
            tableinfo= self.archive[table_ref[1][0]]
            tablemodel= self.archive[tableinfo.tableModel[1][0]]
            if (sheet_msg.name, tablemodel.table_name) != sheet.name:
                continue
            for row in self._row_iter( tablemodel ):
                yield row
            break

def _row_iter( self, tablemodel ):
    str_values= dict()
    for str_ref in tablemodel.data_store.stringTable:
        strs= self.archive[str_ref[1][0]]
        for entry in strs.entries:
            str_values[entry.key]= entry.string
    form_values= dict()
    for form_ref in tablemodel.data_store.formulaTable:
        form= self.archive[form_ref[1][0]]
        for entry in form.entries:
            form_values[entry.key]= entry.formula.AST_node_array.AST_node
    error_values= dict()
    for error_ref in tablemodel.data_store.formulaErrorTable:
        error= self.archive[error_ref[1][0]]
        # TODO:  decode error details
        #for entry in error.entries:
        #    error_values[entry.key]= entry.formula # entry.string?
    for t in tablemodel.data_store.tiles:
        for tt_ref in t.tiles:
            tile= self.archive[tt_ref.ref[1][0]]
            for row in tile.rowInfos:
                yield [ self._cell(row.cells[col], str_values, form_values, error_values)
                    for col in row.cells ]
```

1.8.9 Fixed-Format Workbook

```
import logging
import pprint

from stingray.workbook.base import Workbook
import stingray.sheet
import stingray.cell
```

class workbook.**Fixed_Workbook**

Like a CSV workbook, this is a kind of degenerate case. We don't have a lot of sheets, or a lot of data types.

A subclass might do EBCDIC conversion and possibly even decode packed decimal numbers. To do this, a COBOL-language DDE would be required as the schema definition. See *The COBOL Package*.

```
class Fixed_Workbook( Workbook ):
    """A file with fixed-sized, no-punctuation fields.

    A schema is **required** to parse the attributes.

    The rows are defined as :py:class:`stingray.sheet.LazyRow` instances so that
    bad data can be gracefully skipped over.
    """
    row_class= stingray.sheet.LazyRow

    def __init__( self, name, file_object=None, schema=None ):
        """Prepare the workbook for reading.

        :param name: File name
        :param file_object: Optional file-like object.  If omitted, the named file is opened.
        :param schema: Schema required for processing.
        """
        super().__init__( name, file_object )
        if self.file_obj:
            self.the_file= None
            self.wb= self.file_obj
        else:
            self.the_file = open( name, 'rt' )
            self.wb= self.the_file
        self.schema= schema
```

Fixed_Workbook.**sheet**(*sheet*)

```
def sheet( self, sheet_name ):
    """sheet_type is ignored; it must be an external schema."""
    return stingray.sheet.ExternalSchemaSheet( self, sheet_name, schema=self.schema )
```

We can build eager `sheet.Row` instances for some kinds of flat files. Eager rows, however, don't generalize well to COBOL structures.

Therefore, we must build `sheet.LazyRow` objects here and defer the data type conversion until `workbook.Fixed_Workbook.row_get()`. Or `cobol.COBOL_File.row_get()`, which can be more complex still.

Fixed_Workbook.**rows_of**(*sheet*)

```
def rows_of( self, sheet ):
    """An iterator over all rows of the named sheet.
    For Fixed files, the sheet.name is simply ignored.
    """
    self.sheet= sheet
    for data in self.wb:
        logging.debug( pprint.pformat( data, indent=4 ) )
        row = self.row_class( sheet, data=data )
        yield row
```

Fixed_Workbook.**row_get**(*row*, *attribute*)

```
def row_get( self, row, attr ):
    """Create a :py:class:'cell.Cell' from the row's data."""
    extract= row._state['data'][attr.offset:attr.offset+attr.size]
    return attr.create( extract.rstrip(), self )
```

1.9 The iWork '13 Numbers Modules

There are two digressive modules that are part of reading iWork '13 Numbers files. We could depend on other implementations. Instead, we've provided our own implementation to reduce the dependencies.

It's sensible to use more sophisticated imports in the workbook package to properly handle various kinds of Snappy compression and Protobuf representation implementations. In the same way that the various XML workbooks could be built on any of the Python XML modules.

However, it's somewhat simpler for developers to rely only on XLRD and Stingray.

1.9.1 Snappy Module – Unpacking iWork 13 files.

This is not a full implementation of the Snappy compression protocol. It's a minimal implementation, enough to unpack iWork '13 files.

The iWork '13 use of Snappy

https://github.com/obriensp/iWorkFileFormat

https://github.com/obriensp/iWorkFileFormat/blob/master/Docs/index.md

> "Components are serialized into .iwa (iWork Archive) files, a custom format consisting of a Protobuf stream wrapped in a Snappy stream.
>
> "Snappy Compression
>
> "Snappy is a compression format created by Google aimed at providing decent compression ratios at high speeds. IWA files are stored in Snappy's framing format, though they do not adhere rigorously to the spec. In particular, they do not include the required Stream Identifier chunk, and compressed chunks do not include a CRC-32C checksum.
>
> "The stream is composed of contiguous chunks prefixed by a 4 byte header. The first byte indicates the chunk type, which in practice is always 0 for iWork, indicating a Snappy compressed chunk. The next three bytes are interpreted as a 24-bit little-endian integer indicating the length of the chunk. The 4 byte header is not included in the chunk length.

Snappy

http://en.wikipedia.org/wiki/Snappy_(software)

https://code.google.com/p/snappy/

https://code.google.com/p/snappy/source/browse/trunk/format_description.txt

Implementation

Module docstring.

```
"""Read snappy-compressed IWA files used for Numbers '13 workbooks.

This is a variation on the "official" snappy protocol. The CRC checksums
are not used by iWork '13. This is not a full implementation, just
a decoder for iWork snappy-compressed IWA files.

See https://code.google.com/p/snappy/
"""
```

Some overheads

```
import logging
import sys
```

snappy.**bytes_int**(*seq*)

Function to decode bytes into an integer

```
def bytes_int( seq ):
    """8-bit encoded integer as sequence of 1 to 4 bytes, little-endian.
    """
    shift= 0
    v= 0
    for b in seq:
        v += b<<shift
        shift += 8
    return v
```

snappy.**varint**(*stream*)

Function to decode varint-encoded bytes.

```
def varint( stream ):
    """7-bit encoded integer as a sequence of bytes, little-endian.
    MSB is used to indicate if more bytes are part of this value.

    >>> varint( iter([0xfe, 0xff, 0x7f]) )
    2097150
    """
    b= next(stream)
    shift= 0
    v = (b & 0x7F) # <<shift to be pedantic
    while b & 0x80 != 0:
        b= next(stream)
        shift += 7
        v += (b & 0x7F)<<shift
    return v
```

class snappy.**Snappy**

The snappy decode class has two parts.

- The LZ77 decoder which expands the tags to create the data. There are four kinds of tags.
  - 0b00: literal

    Literals are uncompressed data stored directly in the byte stream. The literal length is stored differently depending on the length of the literal:

    * For literals up to and including 60 bytes in length, the upper six bits of the tag byte contain (len-1). The literal follows immediately thereafter in the bytestream.
    * For longer literals, the (len-1) value is stored after the tag byte, little-endian. The upper six bits of the tag byte describe how many bytes are used for the length; 60, 61, 62 or 63 for 1-4 bytes, respectively. The literal itself follows after the length.
  - 0b01: Copy with 1-byte offset

    These elements can encode lengths between [4..11] bytes and offsets between [0..2047] bytes. (len-4) occupies three bits and is stored in bits [2..4] of the tag byte. The offset occupies 11 bits, of which the upper three are stored in the upper three bits ([5..7]) of the tag byte, and the lower eight are stored in a byte following the tag byte.
  - 0b10: Copy with a 2-byte offset

    These elements can encode lengths between [1..64] and offsets from [0..65535]. (len-1) occupies six bits and is stored in the upper six bits ([2..7]) of the tag byte. The offset is stored as a little-endian 16-bit integer in the two bytes following the tag byte.
  - 0b11: Copy with a 4-byte offset

    These are like the copies with 2-byte offsets (see previous subsection), except that the offset is stored as a 32-bit integer instead of a 16-bit integer (and thus will occupy four bytes).
- The higher-level framing protocol. There is just one kind of frame, type "0" (Compressed Data) frame with a three-byte length.

```
class Snappy:
    def __init__( self ):
        self.log= logging.getLogger( self.__class__.__qualname__ )
```

The LZ77 decoder. This locates the **varint** size header. That's followed by a sequence of tags. The literal tag has data. The other three tags repeat previously output bytes.

Because of the framing protocol, we're limited to a buffer of only 64K bytes.

```
def lz77( self, frame ):
    """Decode one frame of a Snappy LZ77-encoded stream.

    Get the tags, data and emit the resulting uncompressed bytes for this frame.

    There are four types of tags:

    0b00 - Literal - the balance of the tag specifies the length.
    0b01 - Copy 1-byte offset - repeat previous bytes from the output buffer.
    0b10 - Copy 2-byte offset - repeat previous bytes
    0b11 - Copy 4-byte offset - repeat previous bytes

    :param frame: One frame from a Snappy file.
    :returns: buffer of bytes for this frame.
    """
    buffer= bytearray()
    stream= iter( frame )
    # The size of the uncompressed data in this frame.
    size= varint( stream )
    self.log.debug( "  LZ77 size {0}".format(size) )
    # Build the uncompressed buffer.
    while len(buffer) < size:
        hdr= int(next(stream))
        tag_upper, element_type = hdr >> 2, hdr & 0b11

        if element_type == 0b00: # Literal
            if tag_upper < 60:
                size_elt= tag_upper
            else:
                size_elt= bytes_int( next(stream) for i in range(tag_upper - 59) )
            bytes= [next(stream) for b in range(size_elt+1)]
            self.log.debug(
                "{0:08b} {1} {2} = {3!r}".format(
                hdr, element_type, size_elt, bytes) )
            buffer.extend( bytes )
        else: # Some kind of copy
            # Copy -- gather bytes based on offset, stow into buffer based on length
            if element_type == 0b01: # Copy with 1-byte offset
                length, offset_hi = tag_upper & 0b111, (tag_upper & 0b111000)>>3
                offset_lo= next(stream)
                offset= (offset_hi<<8)+offset_lo
                self.log.debug(
                    "{0:08b} {1:8b} {2} {3} = {4!r} {5!r}".format(
                    hdr, offset_lo, element_type, length, (offset_hi, offset_lo), offset) )
                length += 4
            elif element_type == 0b10: # Copy with 2-byte offset
                offset= bytes_int( next(stream) for i in range(2) )
                length= tag_upper
                self.log.debug( "{0:08b} {1} {2} {3}".format(hdr, element_type, length, offset) )
                length += 1
            elif element_type == 0b11: # Copy with 4-byte offset
                offset= bytes_int( next(stream) for i in range(4) )
                length= tag_upper
                self.log.debug(
                    "{0:08b} {1} {2} {3}".format(
                    hdr, element_type, length, offset) )
                length += 1
            else:
                raise Exception( "Logic Problem" )
            # Extend buffer with the copied bytes.
            # Handle RLE feature, if necessary.
            copy= buffer[-offset:]
            if offset < length:
                repeat= copy[:]
                while len(copy) < length:
                    copy += repeat
            buffer.extend( copy[:length] )
    assert len(buffer) == size, "len(buffer) {0} != size {1}".format(len(buffer),size)
    return buffer
```

The Framing protocol required to decode. The frames contain up to 64K of compressed data. This defines a sequence of windows over the stream of data.

```
def decompress( self, file_object ):
    """Decompress a snappy file object. Locate each frame in the snappy
    framing protocol that’s used by iWork (not precisely as specified
    by Google.) For each frame, do the LZ77 expansion on the frame’s bytes to
    build the uncompressed data.

    Frames have a 4-byte header. Byte 0 is frame type, only type 0 (Compressed Data)
    is supported. Bytes 1-3 are a 24-bit size for the frame.
    Practically, it’s limited to 65536 bytes.

    The CRC32 is omitted for iWork files

    ..  todo:: yield iterable byte stream for use in higher-levels of the protocol.

        It’s not *required* to materialize the entire data buffer as a single object.
        The intent of the framing is to limit the size of the buffer required.

    Note that we could provide ``file_object`` file directly to ``lz77()`` function because
    lz77 protocol starts with the target uncompressed size at the front of the frame.
    We don’t **actually** need to read the frame here.
    """
    data= bytearray()
    header= file_object.read(4)
    while header:
        # The Snappy framing format: type 0 (Compressed Data) with a 24-bit size.
        # The CRC32 is omitted for iWork files
        type_frame, size_frame = header[0], bytes_int(header[1:4])
        assert type_frame == 0, "Unsupported Snappy Frame {0}".format(type_frame)
        self.log.debug( "Frame type {0} size {1}".format( type_frame, size_frame ) )
        frame= file_object.read( size_frame )
        data.extend( self.lz77( frame ) )
        header= file_object.read(4)
    return data
```

1.9.2 Protobuf Module – Unpacking iWork 13 files.

This is not a full implementation of Protobuf object representation. It’s a minimal implementation, enough to unpack iWork ‘13 files.

The iWork ‘13 use of protobuf

https://github.com/obriensp/iWorkFileFormat

https://github.com/obriensp/iWorkFileFormat/blob/master/Docs/index.md

> “Components are serialized into .iwa (iWork Archive) files, a custom format consisting of a Protobuf stream wrapped in a Snappy stream.
>
> “Protobuf
>
> “The uncompresed IWA contains the Component’s objects, serialized consecutively in a Protobuf stream. Each object begins with a varint representing the length of the ArchiveInfo message, followed by the ArchiveInfo message itself. The ArchiveInfo includes a variable number of MessageInfo messages describing the encoded Payloads that follow, though in practice iWork files seem to only have one payload message per ArchiveInfo.
>
> “Payload

> "The format of the payload is determined by the type field of the associated MessageInfo message. The iWork applications manually map these integer values to their respective Protobuf message types, and the mappings vary slightly between Keynote, Pages and Numbers. This information can be recovered by inspecting the TSPRegistry class at runtime.
>
> "TSPRegistry"
>
> "The mapping between an object's MessageInfo.type and its respective Protobuf message type must by extracted from the iWork applications at runtime. Attaching to Keynote via a debugger and inspecting [TSPRegistry sharedRegistry] shows:
>
> "A full list of the type mappings can be found here."
>
> https://github.com/obriensp/iWorkFileFormat/blob/master/iWorkFileInspector/iWorkFileInspector/Persistence/MessageTypes

Message `.proto` files.

- Table details

  https://github.com/obriensp/iWorkFileFormat/blob/master/iWorkFileInspector/iWorkFileInspector/Messages/Proto/TSTArchives.

- Numbers details

  https://github.com/obriensp/iWorkFileFormat/blob/master/iWorkFileInspector/iWorkFileInspector/Messages/Proto/TNArchives.p

- Calculating Engine details

  https://github.com/obriensp/iWorkFileFormat/blob/master/iWorkFileInspector/iWorkFileInspector/Messages/Proto/TSCEArchive

- Structure (i.e., TreeNode, perhaps more relevant for Keynote)

  https://github.com/obriensp/iWorkFileFormat/blob/master/iWorkFileInspector/iWorkFileInspector/Messages/Proto/TSKArchives.

We require two of the files from this project to map the internal code numbers

- https://github.com/obriensp/iWorkFileFormat/blob/master/iWorkFileInspector/iWorkFileInspector/Persistence/MessageTypes/Nu
- https://github.com/obriensp/iWorkFileFormat/blob/master/iWorkFileInspector/iWorkFileInspector/Persistence/MessageTypes/Co

These files are incorporated into this module as separate `*.json` files. See *Installation via setup.py* for more information on these files.

protobuf

https://developers.google.com/protocol-buffers/

https://developers.google.com/protocol-buffers/docs/encoding

http://en.wikipedia.org/wiki/Protocol_Buffers

IWA Structure

Each IWA has an ArchiveInfo message.

```
message ArchiveInfo {
    optional uint64 identifier = 1;
    repeated MessageInfo message_infos = 2;
}
```

Within the ArchiveInfo is a MessageInfo message.

```
message MessageInfo {
    required uint32 type = 1;
    repeated uint32 version = 2 [packed = true];
    required uint32 length = 3;
    repeated FieldInfo field_infos = 4;
    repeated uint64 object_references = 5 [packed = true];
    repeated uint64 data_references = 6 [packed = true];
}
```

The MessageInfo is followed by the payload. That must be decoded to get the actual data of interest.

Implementation

Module docstring.

```
"""Read protobuf-serialized messages from IWA files used for Numbers '13 workbooks.

https://developers.google.com/protocol-buffers/

https://developers.google.com/protocol-buffers/docs/encoding

Requires :file:`Numbers.json` and :file:`Common.json` from the installation
directory.
"""
```

Some Overheads

```
import logging
import sys
import os
import json
from collections import defaultdict, ChainMap
from stingray.snappy import varint
```

class `protobuf.`**`Message`**

A definition of a generic protobuf message. This is both an instance and it also as staticmethods that build instances from a buffer of bytes.

We don't use subclasses of `Message`. The proper way to use Protobuf is to compile `.proto` files into Message class definitions.

```
class Message:
    """Generic protobuf message built from sequence of bytes.

    :ivar name_: the protobuf message name.
    :ivar fields: a dict that maps field numbers to field values.
        The contained message objects are **not** parsed, but left as
        raw bytes.
    """
    def __init__( self, name, bytes ):
        self.name_= name
        self.fields= Message.parse_protobuf(bytes)
    def __repr__( self ):
        return "{0}({1})".format( self.name_, self.fields )
    def __getitem__( self, index ):
        return self.fields.get(index,[])
```

An iterative parser for the top-level (name, value) pairs in the protobuf stream. This yields all of the pairs that are parsed.

```
@staticmethod
def parse_protobuf_iter( message_bytes ):
    """Parse a protobuf stream, iterating over the name-value pairs that are present.
    This does NOT recursively descend through contained sub-messages.
    It does only the top-level message.
    """
    bytes_iter= iter(message_bytes)
    while True:
        try:
            item= varint( bytes_iter )
        except StopIteration:
            item= None
            break
        field_number, wire_type = item >> 3, item & 0b111
        if wire_type == 0b000:   # varint representation
            item_size= None # varint sizes vary, need something for debug message
            field_value = varint( bytes_iter )
        elif wire_type == 0b001: # 64-bit == 8-byte
            item_size= 8
            field_value = tuple( next(bytes_iter) for i in range(8) )
        elif wire_type == 0b010: # varint length and then content
            item_size= varint( bytes_iter )
            field_value = tuple( next(bytes_iter) for i in range(item_size) )
        elif wire_type == 0b101: # 32-byte == 4-byte
            item_size= 4
            field_value = tuple( next(bytes_iter) for i in range(4) )
        else:
            raise Exception( "Unsupported {0}: {1}, {2}", bin(item), field_number, wire_type )
        Message.log.debug(
            ’{0:b}, field {1}, type {2}, size {3}, = {4}’.format(
            item, field_number, wire_type, item_size, field_value) )
        yield field_number, field_value
```

Create a bag in the form of a mapping `{name: [value,value,value], ... }`. This will contain the top-level identifiers and the bytes that could be used to parse lower-level messages.

```
@staticmethod
def parse_protobuf( message_bytes ):
    """Creates a bag of name-value pairs. Names can repeat, so values are
    an ordered list.
    """
    bag= defaultdict( list )
    for name, value in Message.parse_protobuf_iter( message_bytes ):
        bag[name].append( value )
    return dict(bag)
```

A class-level logger. We don’t want a logger for each instance, since we’ll create many `Message` instances.

```
Message.log= logging.getLogger( Message.__class__.__qualname__ )
```

class `protobuf.`**`Archive_Reader`**

A Reader for IWA archives. This requires that the archive has been processed by the `snappy.Snappy` decompressor.

```
class Archive_Reader:
    """Read and yield Archive entries from MessageInfo.type and payload.
    Resolves the ID into a protobuf message name.
    Mapping from types to messages

    -   https://github.com/obriensp/iWorkFileFormat/blob/master/iWorkFileInspector/iWorkFileInspecto

    -   https://github.com/obriensp/iWorkFileFormat/blob/master/iWorkFileInspector/iWorkFileInspecto
    """
    def __init__( self ):
        self.tsp_names= self._tsp_name_map()
        self.log= logging.getLogger( self.__class__.__qualname__ )
```

Mapping from internal code numbers of protobuf message class names. This requires `Numbers.json` and `Common.json` from the installation directory.

```
@staticmethod
def _tsp_name_map():
    """Build the TSPRegistry map from messageInfo.type to message proto
    """
    def load_map( filename ):
        """JSON documents have string keys: these must be converted to int."""
        installed= os.path.dirname(__file__)
        with open( os.path.join(installed, filename) ) as source:
            raw= json.load( source )
        return dict( (int(key), value) for key, value in raw.items() )

    tsp_names= ChainMap(
        load_map("Numbers.json"),
        load_map("Common.json"),
    )
    return tsp_names
```

Create the payload message from a MessageInfo instance and the payload bytes.

```
def make_message( self, messageInfo, payload ):
    name= self.tsp_names[messageInfo[1][0]]
    return Message( name, payload )
```

Iterate through all messages in this IWA archive. Locate the ArchiveInfo, MessageInfo and Payload. Parse the payload to create the final message that's associated with the ID in the ArchiveInfo.

```
def archive_iter( self, data ):
    """Iterate through the iWork protobuf-serialized archive:
    Locate the ArchiveInfo object.
    Each ArchiveInfo contains a MessageInfo(s).
    Each MessageInfo describes a payload.
    It appears that there's only one MessageInfo per ArchiveInfo
    even though the ``.proto`` file indicates multiple as possible.

    Yield a sequence of iWork archived messages as pairs:
        identifier, Message object built from the payload.
    """
    protobuf= iter(data)
    while True:
        try:
            size= varint( protobuf )
        except StopIteration:
            size= None
            break
        self.log.debug( "{0} bytes".format(size) )
```

```
message_bytes= [ next(protobuf) for i in range(size) ]
archiveInfo= Message( "ArchiveInfo", message_bytes)
archiveInfo.identifier = archiveInfo[1][0]
archiveInfo.message_infos = [
    Message("MessageInfo", mi) for mi in archiveInfo[2] ]
self.log.debug( " ArchiveInfo identifier={0} message_infos={1}".format(
    archiveInfo.identifier, archiveInfo.message_infos) )
messageInfo_0= archiveInfo.message_infos[0]
messageInfo_0.length= messageInfo_0[3][0]
self.log.debug( "   MessageInfo length={0}".format(messageInfo_0.length) )
payload_raw= [ next(protobuf) for i in range(messageInfo_0.length) ]
self.log.debug( "     Payload {0!r}".format(payload_raw) )
message= self.make_message( messageInfo_0, payload_raw )
yield archiveInfo.identifier, message
```

1.10 The COBOL Package

Or. "How do I access COBOL-defined data from Python"?

We have three problems to solve to get COBOL data into Python applications.

- Most of the `cell.Cell` subclasses aren't really appropriate for data coming from COBOL applications. Indeed, only `cell.TextCell` is really appropriate.
- The schema more complex than the flat-file schema expected by `schema`.
  - The structure is hierarchical.
  - Each attribute has a large number of properties required for proper conversion to Python types.
  - There are repeating groups (based on the "OCCURS" clauses).
  - There are attributes with locations based on other attributes. These are defined by Occurs Depending On clauses.
  - There are fields that may have invalid data due to a "REDEFINES" clause. The USAGE information indicates the encoding of the data which is expected to be in the field, the actual data may not match the definition.
- The schema is encoded in COBOL. That is the subject of *COBOL Loader Module – Parse COBOL Source to Load a Schema*.

1.10.1 Requirements

The essential COBOL use case is to create usable Python objects from the source file data.

For each source file row, there's a two-step operation.

1. Access elements of each row using the COBOL DDE structure.
2. Build Python objects from the Cells found in the row. Building Python objects is best done with a "builder" function, as shown above in *Schema Package – Schema and Attribute Definitions*, *The Stingray Developer's Guide*, and *Stingray Demo Applications*.
3. Split the file into sections for parallel processing. The GNU/Linux `split` command won't work with EBCDIC files, so we have to use the low-level RECFM definitions to parse and split a file.

High-Level Processing

A sheet.Row appears to be a sequential collection of cell.Cell instances. The schema is used for **lazy** creation of cell.Cell instances. Cells may have bad data, and the use of *REDEFINES* means that a row cannot be built eagerly.

```
with open("sample/zipcty.cob", "r") as cobol:
    schema= stingray.cobol.loader.COBOLSchemaLoader( cobol ).load()
    #pprint.pprint( schema )
for filename in 'sample/zipcty1', 'sample/zipcty2':
    with stingray.cobol.Character_File( filename, schema=schema ) as wb:
        sheet= wb.sheet( filename )
        counts= process_sheet( sheet )
        pprint.pprint( counts )
```

The Character_File class is for files in all character (no packed decimal) encoded in ASCII. These kinds of files are expected to have proper '\n' characters at the end of each record.

For an EBCDIC file, use the cobol.EBCDIC_File class. These files lack a specific record delimiter character. If the "RECFM" (Record Format) is V or VB, there's a header word with length instead of a delimiter.

Processing a COBOL "sheet" is a slight extension to processing workbook sheets. We can turned the simple sequential COBOL-based schema into a dictionary. The keys are the simple field names as well as the full field paths.

```
def process_sheet( sheet ):
    # Simple field names
    schema_dict= dict( (a.name, a) for a in sheet.schema )
    # Dot-puncutated field paths
    schema_dict.update( dict( (a.path, a) for a in sheet.schema ) )

    counts= { 'read': 0 }
    row_iter= sheet.rows()
    row= next(row_iter)
    header= header_builder( row, schema_dict )
    print( header )

    for row in row_iter:
        data= row_builder( row, schema_dict )
        print( data )
        counts['read'] += 1
    return counts
```

This relies on two functions to access elements: header_builder() and row_builder(). Each of these will use the dictionary schema to access individual elements.

Access Elements Via Schema

A function like the following builds an object from a row using the schema.

```
def header_builder(row, schema):
    return types.SimpleNamespace(
        file_version_year= row.cell(schema['FILE-VERSION-YEAR']).to_str(),
        file_version_month= row.cell(schema['FILE-VERSION-MONTH']).to_str(),
        copyright_symbol= row.cell(schema['COPYRIGHT-SYMBOL']).to_str(),
        tape_sequence_no= row.cell(schema['TAPE-SEQUENCE-NO']).to_str(),
    )
```

We use schema['FILE-VERSION-YEAR'] to track down the attribute details. We use this information to build a Cell from the current row.

We also need to handle repeating groups. In the case of repeating elements, the repeating collection of elements creates a tuple-of-tuples structure which we can index.

This gives us two possible ways to provide an OCCURS index values. The first possibility is to associate the index with the `schema.Attribute`.

```
def process_sheet( sheet ):
    for row in sheet.rows():
        foo= row.cell(schema[n]).to_str()
        bar= row.cell(schema[m].index(i, j)).to_str()
```

While the above is weird, it parallels COBOL usage of indexing and element naming. The idea is to provide the lowest level of name and all the indexes above it in one construct.

This is appealing because we can use the existing `sheet.Row.cell()` method without any modification. The downside of this is that we're creating a tweaked attribute definition, which is a terrible idea. On balance, the use of `schema.Attribute.index()` has to be rejected.

The second way to provide an index is provide a subset of the indexes and get a kind of slice. This is like the positional version, shown above.

```
def process_sheet( sheet ):
    for row in sheet.rows():
        foo= row.cell(schema[n]).to_str()
        bar= row.cell(schema[m])[i][j].to_str()
```

Possible Added Fluency

The path names are awkward to use since they must include every level from the `01` to the relevant item.

A subclass of `schema.Schema` *could* introduce a method like `get_name()` to make a schema into a slightly more fluent mapping in addition to a sequenmce.

```
foo= row.cell(schema_dict.get_name(‘bar’))
baz_i_j= row.cell(schema_dict.get_name(‘baz’).index(i, j))
```

This doesn't work for non-unique names.

The COBOL `OF` syntax *could* be modeled using a fluent `of()` method.

```
foo= row.cell(schema_dict.get(‘foo’).of(‘bar’))
baz_i_j= row.cell(schema_dict.get(‘baz’).index(i, j))
baz_i_quux_j = row.cell(schema_dict.get(‘baz’).index(i).of(‘quux’).index(j))
```

This is all potentially useful for hyper-complex COBOL record layouts.

Incremental index calculation involves creating an interim, stateful Attribute definition. The "cloning" and "tweaking" of an `schema.Attribute` definition is a bad design. The index processing is stateful, and we would need to accumulate index information somehow along the path of fluent methods.

This works out better if the `sheet.Row.cell` method handles the index calculations entirely separate from the attribute navigation. The following seems like a more sensible way to handle this.

```
foo= row.cell(schema_dict.get(‘foo’).of(‘bar’))
baz_i_j= row.cell(schema_dict.get(‘baz’), i)
baz_i_quux_j = row.cell(schema_dict.get(‘baz’).of(‘quux’), (i, j))
```

The first example provides no index, a multi-dimensional sequence of *Cell* objects is returned.

The second example provides too few indices, a sequence of *Cell* objects is returned.

The third example provides all indices, an individual *Cell* is returned.

Generic Processing

The `cobol.dump()` function can dump a record showing raw bytes.

This relies on the `cobol.dump_iter()` function. This function iterates through all DDE elements (in order) providing a five-tuple of the original DDE, the derived attribute, a tuple of specific index values used, the raw bytes found and any `Cell` object if the data was actually valid.

The indices tuple may be an empty tuple for non-OCCURS data. For OCCURS data, all combinations of index values are used, so large, nested structures may produce long lists of values.

The raw bytes and the `Cell` object are (technically) redundant, since all subclasses of `Cell` used by the `cobol` package have a `raw` attribute with the raw bytes. However, it's sometimes simpler to have this expanded in the tuple.

The Occurs Depending On Problem

There are two kinds of DDE's: group and elementary. A group DDE is a collection of DDE's. An elementary does not contain anything. This makes a DDE a proper tree.

A DDE, d, has two interesting properties: the offset, d_o, and the size, d_s. We're interested in the bytes associated with a particular DDE element, $B[d]$. These bytes are fetched from a larger buffer, $B = \{B_0, B_1, B_2, ..., B_n\}$.

This allows us to fetch the bytes from a buffer by getting bytes $B[d] = \{B_x | d_o \leq x < d_o + d_s\}$.

The size of an elementary DDE, d_s, is fixed by the picture clause.

The size of a group DDE is the sum of the children. $d_s = \sum_{c \in d} c_s$. While this seems clear, it doesn't include the `OCCURS` clause issues: we'll return to those below.

The DDE's in a tree can have three species of relationships.

- Predecessor/Successor. The predecessor of d is $P(d)$.

  The offset of an item is $d_o = P(d)_o + P(d)_s$. This applies recursively to the first item in the DDE collection. For the first item in a DDE, d_0, $P(d_0)_o = 0$ and $P(d_0)_s = 0$.

- Parent/Child. The parent contains a group of items; the parent of d is $G(d)$, we can say $d \in G(d)$.
- Redefines. This DDE's offset (and size) is defined by another item, $R(d)$. $d_o = R(d)_o$ and $d_s = R(d)_s$.

The predecessor/successor relationship is implied by the order of the DDE's as they're compiled. If they have the same level number, they're successors.

The child relationship is specified via the level numbers. A larger level number implies a child of a lower level number.

The redefines relationship is specified via the `REDEFINES` clause.

This is complicated by the `OCCURS` clause. There are two versions in the COBOL language manual, to which we'll add third.

- "Format 1" has a fixed number of occurrences, $O(d) = n$, comes from `OCCURS n TIMES`.
- "Format 2" means the number of occurrences depends on a piece of data, in another DDE, the depends-n basis, $D(d)$. This means that $O(d) = B[D(d)]$. This comes from `OCCURS n TO m TIMES DEPENDING ON b`. The lower and upper bounds, *n* and *m*, are irrelevant. Only the dependency matters.
- To this, we can add "Format 0" which has a single occurrence, $O(d) = 1$. This is the default if no occurrence information is provided.

This adds another interesting attribute, the total size of an item, $d_t = d_s \times O(d)$. The total size is the elementary size times the number of occurrences.

This changes our definition of size of a group item to the sum of the total sizes, not the sum of the simple size. $d_t = d_s = \sum_{c \in d} c_t \times O(d)$.

This also changes how we fetch the bytes, $B[d]$, because we need index information for each `OCCURS` clause in the parents of *d*. In COBOL we might be getting `D( I )`.

$$B[d:i] = \{B_x | d_o + (i)d_s \leq x < d_o + (i+1)d_s\}$$

The indices may be much more complex, however. A common situation is a group-level item with an occurrence that nests an elementary item with an occurrence. This is a two-dimensional structure. In COBOL it might be `D( J ) OF G( I )`. More commonly, it's written `D( I, J )` where the indices are applied from top-most group DDE down the tree to the lowest-level items.

$$B[d:i,j] = \{B_x | G(d:i)_o + jd_s \leq x < G(d:i)_o + (j+1)d_s\}$$

The offset, d_o is computed recursively using a combination of predecessors, $P(d)$, groups, $G(d)$, and redefinitions, $R(d)$. There are several cases.

- If there's a REDEFINES clause, $d_o = R(d)_o$.
- If there's a predecessor, $d_o = P(d)_o + P(d)_s \times O(P(d))$.
- If there's no predecessor, there may be a containing group, $d_o = G(d)_o + G(d)_s \times O(G(d))$. This is more complex than it appears because a group could contain an occurrence clause. $G(d:i)_o = G(d)_o + iG(d)_s$.
- If there's no predecessor and no group, $d_o = 0$.

The two cases which involve the occurs information, $O(P(d))$ and $O(G(d))$ may include "format 2" occurs and depend on data within an actual record. We may have $O(P(d)) = B[D(P(d))]$ or $O(G(d)) = B[D(G(d))]$.

Variable-Length COBOL Records

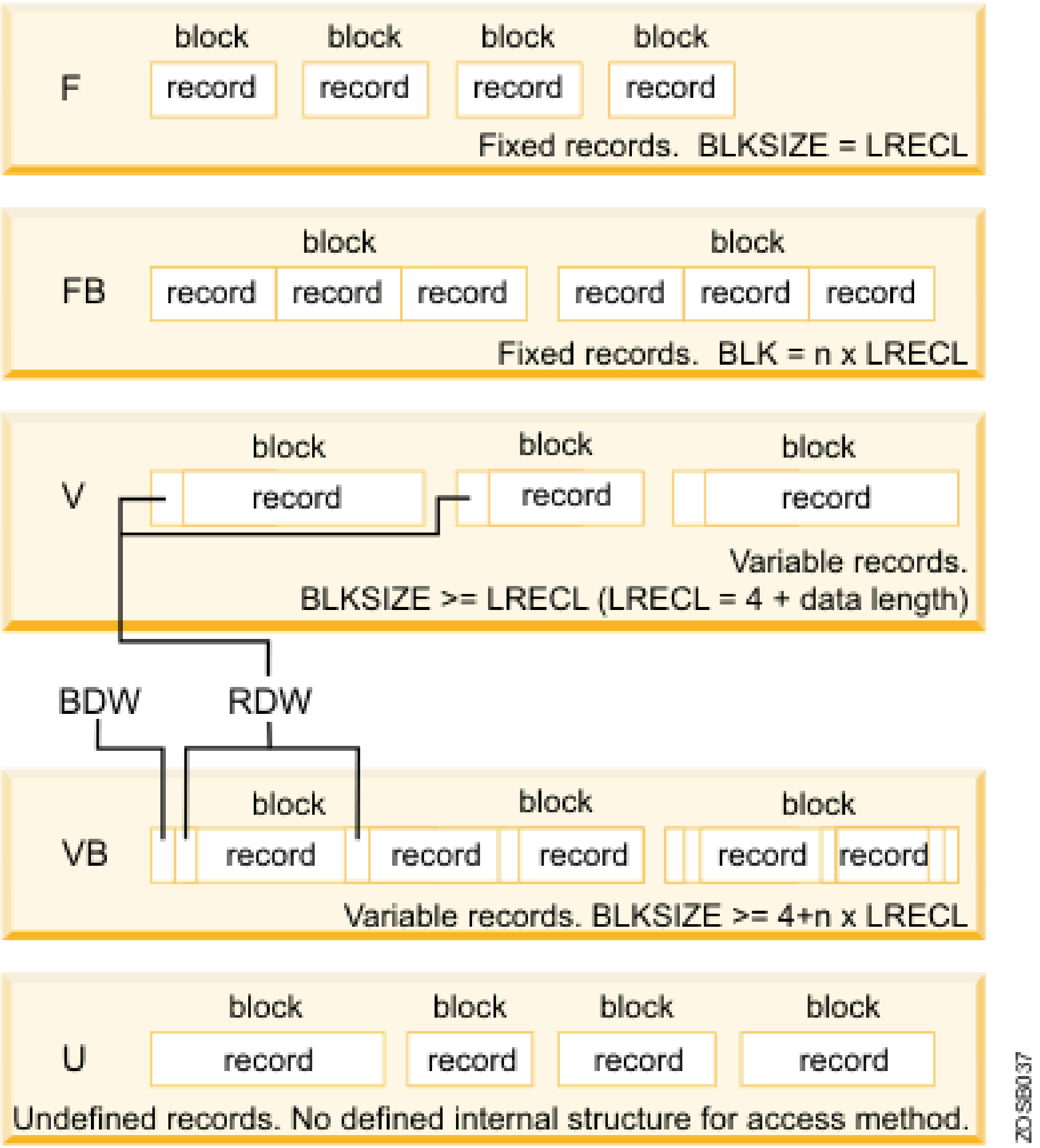
https://publib.boulder.ibm.com/infocenter/zos/basics/index.jsp?topic=/com.ibm.zos.zconcepts/zconcepts_159.htm

V (Variable)

This format has one logical record as one physical block. A variable-length logical record consists of a record descriptor word (RDW) followed by the data. The record descriptor word is a 4-byte field describing the record. The first 2 bytes contain the length of the logical record (including the 4-byte RDW). The length can be from 4 to 32,760 bytes. All bits of the third and fourth bytes must be 0, because other values are used for spanned records. This format is seldom used.

VB (Variable Blocked)

This format places several variable-length logical records (each with an RDW) in one physical block. The software must place an additional Block Descriptor Word (BDW) at the beginning of the block, containing the total length of the block.


F
block
record
Fixed records. BLKSIZE = LRECL
FB
Fixed records. BLK = n x LRECL
V
Variable records.
BLKSIZE >= LRECL (LRECL = 4 + data length)
BDW
RDW
VB
Variable records. BLKSIZE >= 4+n x LRECL
U
Undefined records. No defined internal structure for access method.
ZOS8037

See http://en.wikipedia.org/wiki/Data_set_(IBM_mainframe)

See http://www.simotime.com/vrecex01.htm

There are three relevant encoding possibilities. These are the COBOL "RECFM" options in the JCL for the file. Each record is preceded by a Record Descriptor Word (RDW).

V Variable.

The four bytes preceding the logical record is the Record Descriptor Word. The content is as follows.

Bytes	Description
1-2	This is the length of the logical record plus the length of the four-byte Descriptor Word.
3-4	Usually low values

VB Variable Blocked.

The four bytes preceding the logical record (Descriptor Word) are as follows.

Bytes	Description
1-2	This is the length of the logical record plus the length of the four-byte Descriptor Word.
3-4	The length of the block including four-byte Descriptor Word.

A block can have multiple records in it. The block length must be >= record length.

`VBS` Variable Blocked Spanned.

The four bytes preceding the logical record is the Segment Descriptor Word. The content is as follows.

Bytes	Description
1-2	This is the length of the logical record plus the length of the four-byte Descriptor Word.
3	Segment Control Codes: see below.
4	Low value, reserved for future use

Segment Control Code

Value	Relative position of segment
00	A complete logical record
01	The first segment of a multiple segments record
02	The last segment of a multiple segments record
03	A middle segment of a multiple segments record

This RECFM detail must be provided as part of opening the workbook/sheet so that rows can be properly located within the content.

We've added a series of RECFM classes as a **Strategy** to read files with variable length records.

Low-Level Split Processing

We may have a need to split an EBCDIC file, similar to the Posix `split` command. This is done using `cobol.RECFM` parsers to read records and write to new file(s).

A splitter looks like this:

```
import itertools
import stringray.cobol
import collections
import pprint

batch_size= 1000
counts= collections.defaultdict(int)
with open( "some_file.schema", "rb" ) as source:
    reader= stringray.cobol.RECFM_VB( source ).bdw_iter()
    batches= itertools.groupby( enumerate(reader), lambda x: x[0]//batch_size ):
    for group, group_iter in batches:
        with open( "some_file_{0}.schema".format(group), "wb" ) as target:
        for id, row in group_iter:
            target.write( row )
            counts['rows'] += 1
            counts[str(group)] += 1
pprint.pprint( dict(counts) )
```

There are several possible variations on the construction of the `reader` object.

- cobol.RECFM_F(source).record_iter() – result is RECFM_F.
- cobol.RECFM_F(source).rdw_iter() – result is RECFM_V; RDW's have been added.
- cobol.RECFM_V(source).rdw_iter() – result is RECFM_V; RDW's have been preserved.

- cobol.RECFM_VB(source).rdw_iter() – result is RECFM_V; RDW's have been preserved; BDW's have been discarded.
- cobol.RECFM_VB(source).bdw_iter() – result is RECFM_VB; BDW's and RDW's have been preserved. The batch size is the number of blocks, not the number of records.

The Bad Data Problem

Even with a `sheet.LazyRow`, we have to be tolerant of COBOL data which doesn't match the schema. The `cobol.defs.Usage.create_func()` function may encounter an exception. If so, then a `cobol.defs.ErrorCell` is created instead of the class defined by the Usage and Picture classes.

The `cobol.defs.ErrorCell` includes the raw bytes, but a value of `None`.

1.10.2 Implementation

These are the modules that extend the core functionality of Stingray to handle COBOL files and EBCDIC data.

COBOL Package – Extend Schema to Handle EBCDIC

The COBOL package is a (large) Python `__init__.py` module which includes much of the public API for working with COBOL files.

This module extends Stingray in several directions.

- A new `schema.Attribute` subclass, `cobol.RepeatingAttribute`.
- A handy `cobol.dump()` function.
- The hierarchy of classes based on `cobol.COBOL_File` which provide more sophisticated COBOL-based workbooks.

Within the package we have the `cobol.loader` module which parses DDE's to create a schema.

Module Overheads

We depend on `cell`, `schema`, and `workbook`. We'll also import one class definition from `cobol.defs`.

```
"""stingray.cobol -- Extend the core Stingray definitions to handle COBOL
DDE's and COBOL files, including packed decimal and EBCDIC data.
"""
import codecs
import struct
import decimal
import warnings
import pprint
import logging

import stingray.schema
import stingray.sheet
from stingray.workbook.fixed import Fixed_Workbook


from stingray.cobol.defs import TextCell
```

RepeatingAttribute Subclasses of Attribute

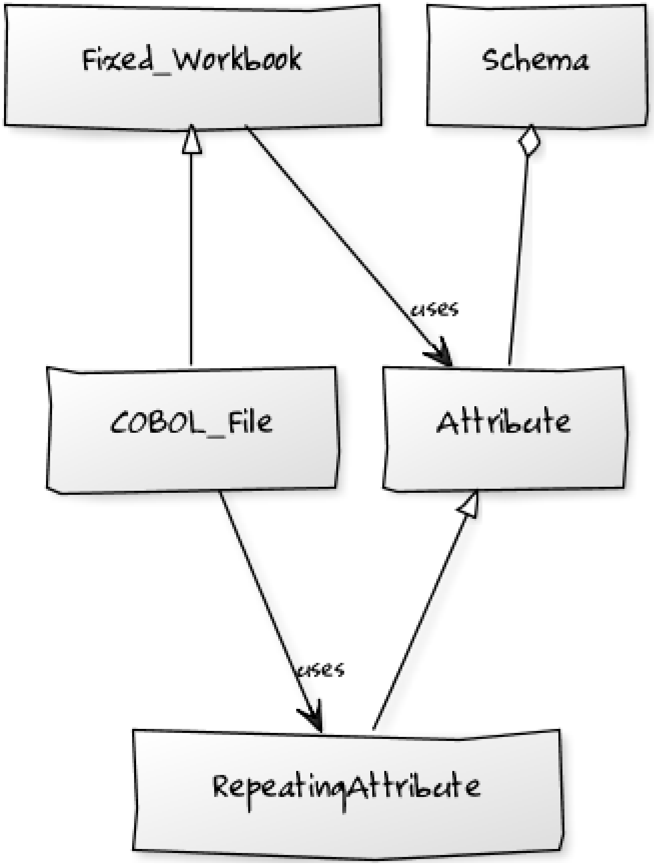
Two new schema.Attribute subclasses are required to carry all the additional attribute information developed during COBOL DDE parsing.

An attribute that has an OCCURS clause (or who's parent has an OCCURS clause) can accept an cobol.RepeatingAttribute.index() method to provide index values used to compute effective offsets.

There are two variants.

- The initial, immutable, cobol.RepeatingAttribute as parsed.
- A working cobol.IndexedAttribute. This is a subclass of cobol.RepeatingAttribute and it contains partial or complete indexing. Partial indexing means that a tuple is built by cobol.COBOL_File.row_get(). Full indexing means that a single Cell can be built.

```
http://yuml.me/diagram/scruffy;/class/
#cobol.attribute,
[Attribute]^[RepeatingAttribute],
[Schema]<>-[Attribute],
[Fixed_Workbook]-uses->[Attribute],
[Fixed_Workbook]^[COBOL_File],
[COBOL_File]-uses->[RepeatingAttribute].
```



In order to fetch data for an ODO OCCURS element, the attribute offsets and sizes cannot **all** be computed during parsing. They must be computed lazily during data fetching. The ODO_LazyRow class handles the Occurs Depending On situation.

Here are the attributes inherited from schema.Attribute.

name The attribute name. Typically always available for most kinds of schema.

create Cell class to create. If omitted, the class-level `Attribute.default_cell` will be used. By default, this refers to `cell.TextCell`.

position Optional sequential position. This is set by the `schema.Schema` that contains this object.

The additional values commonly provided by simple fixed format file schemata. These can't be treated as simple values, however, since they're clearly changed based on the ODO issues.

size Size within the buffer.

These two properties can be tweaked by the `index()` method. If left alone, they simply a delegation to the DDE. If `index()` is used, a subclass object is built where these values come from the `index` method results.

dimensionality A tuple of DDE's that defines the dimensionality pushed down to this item through the COBOL DDE hierarchy.

This meay be set by the `index()` method.

offset Optional offset into a buffer. This may be statically defined, or it may be dynamic because of variably-located data supporting the Occurs Depends On.

This meay be set by the `index()` method.

This subclass introduces yet more attribute-like properties that simply delegate to the DDE.

dde A weakref to a `cobol.loader.DDE` object.

path The "."-separated path from top-level name to this element's name.

usage The original DDE.usage object, an instance of `cobol.defs.Usage`

redefines The original DDE.allocation object, an instance of `cobol.loader.Allocation`

picture The original DDE.picture object, an instance of `cobol.defs.Picture`

size_scale_precision The original DDE.sizeScalePrecision object, a tuple with size, scale and precision derived from the picture.

class `cobol.`**`RepeatingAttribute`**

```
class RepeatingAttribute( stingray.schema.Attribute ):
    """An attribute with dimensionality. Not all COBOL items repeat.

    An "OCCURS" clause will define repeating values.
    An "OCCURS DEPENDING ON" clause may define variably located values.
    """
    default_cell= TextCell
    def __init__(self, name, dde, offset=None, size=None, create=None, position=None, **kw):
        self.dde= dde
        self.name, self.size, self.create, self.position = name, size, create, position
        if not self.create:
            self.create= self.default_cell
        if offset is not None:
            warnings.warn( "Offset {0} is ignored; {1} used".format(offset, self.dde().offset), stac
        self.__dict__.update( kw )
    def __repr__( self ):
        dim= ", ".join( map( repr, self.dimensionality ) )
        return "Attribute( name={0.name!r}, position={0.position}, offset={0.offset}, size={0.size}, 
            self, dim )
```

`RepeatingAttribute.`**`index`**`(`**values*`)`

If the number of index values matches the dimensionality, we'll return a tweaked attribute which has just the offset required and a dimensionality of `tuple()`.

If the number of index values is insufficient, we'll return a tweaked attribute with which has the starting offset and the dimensions left otherwise unspecified.

If the number of index values is excessive, we'll attempt to pop from an empty list.

Note that py:meth:*index* is applied incrementally when the application supplies some of the indices.

- First, the application supplies some of the indices, creating a tweaked `cobol.RepeatingAttribute` with an initial offset.
- Second, the `COBOL_File` supplies the remaining indices, creating yet more temporary `cobol.RepeatingAttribute` based on the initial offset.

```
def index( self, *values ):
    """"Apply possibly incomplete index values to an attribute.
    We do this by cloning this attribute and setting a modified
    dimensionality and offset.

    :param values: 0-based index values.  Yes, legacy COBOL language is 1-based.
        For Python applications, zero-based makes more sense.
    :returns: A :py:class:'cobol.IndexedAttribute' copy, with modified offset
    and dimensionality that can be used with :py:meth:'COBOL_File.row_get'.
    """
    assert values, "Missing index values"
    # Previosly tweaked Attribute? Or originals?
    offset= self.offset
    dim_list= list(self.dimensionality)
    # Apply given index values.
    val_list= list(values)
    while val_list:
        index= val_list.pop(0)
        dim= dim_list.pop(0)
        offset += dim.size * index
    # Build new subclass object with indexes applied.
    clone= IndexedAttribute( self, offset, dim_list )
    return clone
```

With this, a `row.cell(schema.get('name').index(i))` will compute a proper offset.

We "clone" the attribute to assure that each time we apply (or don't apply) the index, nothing stateful will have happened to the original immutable attribute definition.

Note that an incomplete set of index values forces the underlying workbook to create a Python tuple (or tuple of tuples) structure to contain all the requested values. See `cobol.COBOL_File.row_get()`.

The additional properties which are simply shortcuts so that a generic `cobol.RepeatingAttribute` has access to the DDE details.

```
@property
def dimensionality(self):
    """tuple of parent DDE's. Baseline value; no indexes applied."""
    return self.dde().dimensionality
@property
def offset(self):
    """Baseline value; no indexes applied."""
    return self.dde().offset
@property
def path(self):
    return self.dde().pathTo()
```

```
@property
def usage(self):
    return self.dde().usage
@property
def redefines(self):
    return self.dde().allocation
@property
def picture(self):
    return self.dde().picture
@property
def size_scale_precision(self):
    return self.dde().sizeScalePrecision
```

This is a subclass with (some) indices applied. Since this inherits the `cobol.RepeatingAttribute.index()` method, we can apply indices incrementally.

This is not built directly, but only created by `cobol.RepeatingAttribute.index()` with some (or all) indices applied.

```
class IndexedAttribute( RepeatingAttribute ):
    """An attribute with dimensionality and indexes applied.
    This must be built from a :py:class:`cobol.RepeatingAttribute`. It will copy
    some attributes in an effort to somewhat improve efficiency.
    """
    default_cell= TextCell
    def __init__(self, base, offset, dimensionality ):
        self.dde= base.dde
        self.name, self.size, self.create, self.position = base.name, base.size, base.create, base.po
        self._offset= offset
        self._dimensionality= dimensionality
    @property
    def dimensionality(self):
        """tuple of DDE's; Set by ``attribute.index()``."""
        return self._dimensionality
    @property
    def offset(self):
        """Set by ``attribute.index()``."""
        return self._offset
```

COBOL LazyRow

The `sheet.LazyRow` class is blissfully unaware of the need to compute sizes and offsets for COBOL.

class `cobol.`**`ODO_LazyRow`**

This subclass of `sheet.LazyRow` to provide add the feature to recompute sizes and offsets in the case of a variable-located DDE due to an Occurs Depending On.

```
class ODO_LazyRow( stingray.sheet.LazyRow ):
    """If the DDE is variably-located, tweak the sizes and offsets."""

    def __init__( self, sheet, **state ):
        """Build the row from the bytes.

        :param sheet: the containing sheet.
        :param **state: worksheet-specific state value to save.
        """
        super().__init__( sheet, **state )
        for dde in self.sheet.schema.info.get('dde',[]):
            if dde.variably_located:
                dde.setSizeAndOffset(self)
            self._size= dde.totalSize
        else:
            self._size= len(self._state['data'])
```

Dump a Record

cobol.**dump_iter**()

To support dumping raw data from a record, this will iterate through all items in an original DDE. It will a five-tuple with (dde, attribute, indices, bytes, Cell) for each DDE.

If the DDE does not have an OCCURS clause, the indices will be an empty tuple. Otherwise, each individual combination will be yielded. For big, nested tables, this may turn out to be a lot of combinations.

The bytes is the raw bytes for non-FILLER and non-group elements.

The Cell will be a Cell object, either with valid data or an cobol.defs.ErrorCell.

```
def dump_iter( aDDE, aRow ):
    """Yields iterator over tuples of (dde, attribute, indices, bytes, Cell)"""
    def expand_dims( dimensionality, partial=() ):
        if not dimensionality:
            yield partial
            return
        top = dimensionality[0]
        rest= dimensionality[1:]
        for i in range(top):
            for e in expand_dims( rest, partial+(i,) ):
                yield e
    attr= aDDE.attribute() # Final size and offset details
    if aDDE.dimensionality:
        for indices in expand_dims( aDDE.dimensionality ):
            yield aDDE, aDDE.attribute, indices, aRow.cell(attr,indices).raw, aRow.cell(attr,indices)
    elif aDDE.picture and aDDE.name != "FILLER":
        yield aDDE, aDDE.attribute(), (), aRow.cell(attr).raw, aRow.cell(attr)
    else: # FILLER or group level without a picture: no data is available
        yield aDDE, aDDE.attribute, (), None, None
    for child in aDDE.children:
        #pprint.pprint( child )
        for details in dump_iter( child, aRow ):
            yield details
```

cobol.**dump**()

Dump data from a record, driven by the original DDE structure.

```
def dump( schema, aRow ):
    print( "{:45s} {:3s} {:3s} {!s} {!s}".format("Field", "Pos", "Sz", "Raw", "Cell" ) )
    for record in schema.info['dde']:
        for aDDE, attr, indices, raw_bytes, cell in dump_iter(record, aRow):
            print( "{:45s} {:3d} {:3d} {!r} {!s}".format(
                aDDE.indent*'  '+str(aDDE), aDDE.offset, aDDE.size,
                raw_bytes, cell) )
```

COBOL "Workbook" Files

A COBOL file is – in effect – a single-sheet workbook with an external schema. It looks, then, a lot like `workbook.Fixed_Workbook`.

- A pure character file, encoded UNICODE characters in some standard encoding like UTF-8 or UTF-16. This cannot include COMP or COMP-3 fields because the codec would make a mess of the bit patterns.
- An EBCDIC-encoded byte file. This can include COMP or COMP-3 fields.
- An ASCII-encoded byte file. This can include COMP or COMP-3 fields. While this may exist, it seems to be very rare. We don't implement it.

Note that each cell creation involves two features. This leads to a kind of **Double Dispatch** algorithm.

- The cell type. `cobol.defs.TextCell`, `cobol.defs.NumberDisplayCell`, `cobol.defs.NumberComp3Cell` or `cobol.defs.NumberCompCell`.
- The workbook encoding type. Character or EBCDIC (or ASCII).

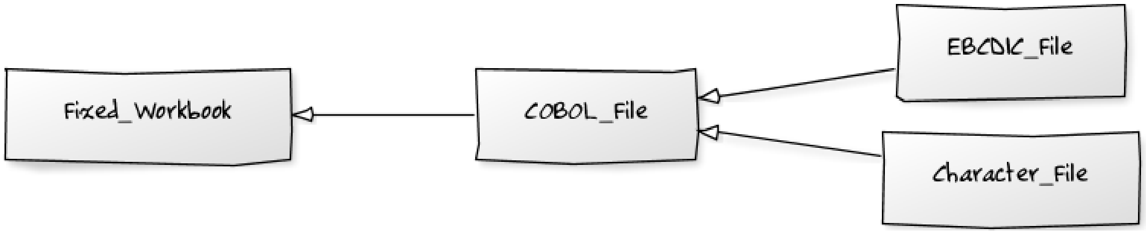
The issue here is we're stuck with a complex "double-dispatch" problem. Each workbook subclass needs to implement methods for `get_text`, `number_display`, `number_comp` and `number_comp3`.

The conversions, while tied to the workbook encoding, aren't properly tied to stateful sheet and row processing in the workbook. They're just bound to the encoding. Consequently, we can make them static methods, possibly even making this a mixin strategy.

The use case looks like this.

1. The application uses `row.cell( schema[n] )`. The `cell()` method is simply `sheet.workbook.row_get( buffer, attribute )`. It applies the cell type (via the schema item's attribute) and the raw data in the row's buffer.
2. `row_get( buffer, attribute )` has to do the following.
   - Convert the buffer into a proper value based on the `attribute` type information **and** the worksheet-specific methods for unpacking the various types of data. The various `cobol` Cell subclasses can refer to the proper conversion methods.
   - Create the required `cell.Cell` based on the `attribute.create(sheet, value)` function.

```
http://yuml.me/diagram/scruffy;/class/
#cobol,
[Fixed_Workbook]^[COBOL_File],
[COBOL_File]^[Character_File],
[COBOL_File]^[EBCDIC_File].
```



COBOL File

class `cobol.`**`COBOL_File`**

This class introduces the expanded version of `row_get` that honors a schema attribute with dimensionality.

```
class COBOL_File( Fixed_Workbook ):
    """A COBOL "workbook" file which uses  :py:class:`cobol.RepeatingAttribute` and
    creates COBOL Cell values.  This is an abstraction which
    lacks specific decoding methods.

    This is a :py:class:`Fixed_Workbook`: a file with fixed-sized, no-punctuation fields.
    A schema is required to parse the attributes.

    The rows are defined as :py:class:`ODO_LazyRow` instances so that
    bad data can be gracefully skipped over and Occurs Depending On offsets
    can be properly calculated.
    """
    row_class= ODO_LazyRow
```

`COBOL_File.`**`row_get_index`**(*row*, *attr*, **index*)

Returning a particular Cell from a row, however, is more interesting for COBOL because the Attribute may contains an “OCCURS” clause. In which case, we may need to assemble a tuple of values.

If there is dimensionality, then take the top-level dimension (`dim[0]`) and use it as an iterator to fetch data based on the rest of the dimensions (`dim[1:]`).

This can assemble a recursive tuple-of-tuples if there are multiple levels of dimensionality.

If too few index values are provided, a tuple of results is built around the missing values.

If enough values are provided, a single result object will be built.

Important: Performance

This is the most-used method. Removing the if-statement would be a huge improvement.

```
def row_get_index( self, row, attr, *index ):
    """Emit a nested-tuple structure of Cell values using the given index values.
    :param row: the source Row.
    :param attr: the  :py:class:`cobol.RepeatingAttribute`; possibly tweaked to
        have an offset and partial dimensions. Or possibly the original tuple
        of dimensions.
    :param index: optional tuple of index values to use.
        Instead of ``row_get( schema.get('name').index(i) )``
        we can use ``row_get_index( schema.get('name'), i )``
    :returns: a (possibly nested) tuple of Cell values matching the dims that lacked
        index values.
    """
    if attr.dimensionality and index:
        # ``attr.index()`` probably not previously used.
        # Apply all remaining values and get the resulting item.
        final= attr.index( *index )
        return self.row_get( row, final )
    elif attr.dimensionality:
        # ``attr.index()`` previously used with partial arg values.
        # Build composite result.
        d= attr.dimensionality[0].occurs.number(row)
        result= []
        for i in range(d):
            sub= attr.index(i)
            result.append( self.row_get( row, sub ) )
        return tuple(result)
    else:
        # Doesn't belong here, delegate.
        return self.row_get( row, attr )
```

COBOL_File.**row_get**(*row*, *attr*)

The API method will get data from a row described by an attribute. If the attribute has dimensions, then indices are used or multiple values are returned by cobol.COBOL_File.row_get_index().

If the attribute is has no dimensions, then it's simply pulled from the source row.

Important: Performance

This is the most-used method. Removing the if-statement would be a huge improvement.

```
def row_get( self, row, attr ):
    """Create a Cell(s) from the row's data.
    :param row: The current Row
    :param attr: The desired Attribute; possibly tweaked to
        have an offset and partial dimensions. Or possibly the original.
    :returns: A single Cell or a nested tuple of Cells if indexes
        were not provided.
    """
    if attr.dimensionality:
        return self.row_get_index( row, attr )
    else:
        extract= row._state['data'][attr.offset:attr.offset+attr.size]
        return attr.create( extract.rstrip(), self, attr=attr )
```

Note that this depends on the superclass, which depends ordinary Unicode/ASCII line breaks. This will not work for EBCDIC files, which may lack appropriate line break characters. For that, we'll need to use specific physical format parsing helpers based on the Z/OS RECFM parameter used to define the file.

Character File This is subclass of COBOL_File that handles COBOL data parsing where the underlying file is text. Since the file is text, Python handles any OS-level bytes-to-text conversions.

class cobol.**Character_File**

```
class Character_File( COBOL_File ):
    """A COBOL "workbook" file with decoding functions for
    proper character data.
    """
```

The following functions are used to do data conversions for COBOL Character files. Text is easy, Python's io.open has already handled this.

```
@staticmethod
def text( buffer, attr ):
    """Extract a text field's value."""
    return buffer
```

Numeric data with usage DISPLAY requires handling implicit decimal points.

```
@staticmethod
def number_display( buffer, attr ):
    """Extract a numeric field's value."""
    final, alpha, length, scale, precision, signed, dec_sign = attr.size_scale_precision
    try:
        if precision != 0:
            if dec_sign == '.' or precision == 0:
                display= buffer
                return decimal.Decimal( buffer )
            else: # dec_sign == "V" or None
                # Insert the implied decimal point.
                display= buffer[:-precision]+"."+buffer[-precision:]
                return decimal.Decimal( display )
        else: # precision == 0:
            display= buffer
            return decimal.Decimal( buffer )
    except Exception:
        Character_File.log.debug( "Can't process {0!r} from {1!r}".format(display,buffer) )
        raise
```

COMP-3 in proper character files may not make any sense at all. A codec would make a hash of the bit patterns required. However, we've defined the method here so that it can be used by the EBCDIC subclass trivially.

We're going to build an ASCII version of the number by decoding the bytes into a mutable bytearray and decorating them with decimal point and sign. This is demonstrably faster and avoids object creation to the extent possible.

```
@staticmethod
def unpack( buffer ):
    """Include ' ' position for leading sign character.
    Trailing sign field will be 48+0xd for negative.
    48+0xf is "unsigned" and 48+0xc is positive.
    """
    yield 32 # ord(b' ')
    for n in buffer:
        yield 48+(n>>4) # ord(b'0')
        yield 48+(n&0x0f)

@staticmethod
def number_comp3( buffer, attr ):
    """Decode comp-3, packed decimal values.

    Each byte is two decimal digits.

    Last byte has a digit plus sign information: 0xd is <0, 0xf is unsigned, and 0xc >=0.
    """
    final, alpha, length, scale, precision, signed, dec_sign = attr.size_scale_precision
    #print( repr(buffer), "from", repr(display) )
    digits = bytearray( Character_File.unpack( buffer ) )
    # Proper sign in front; replace trailing sign with space.
    digits[0]= 45 if digits[-1]==48+0xd else 32 # ord(b'-'), ord(b' ')
    digits[-1]= 32 # ord(' ')
    # Add decimal place if needed.
    if precision:
        digits[-precision:]= digits[-precision-1:-1] # Shift digits to right.
        digits[-precision-1]= 46 # Insert ord(b'.')
    try:
        return decimal.Decimal( digits.decode("ASCII") )
    except Exception:
        Character_File.log.debug( "Can't process {0!r} from {1!r}".format(digits,buffer) )
        raise
```

COMP in proper character files may not make any sense, either. A codec would make a hash of the bit patterns required. Gagin, we've defined it here because that's relatively simple to extend.

We're simply going to unpack big-ending bytes.

```
@staticmethod
def number_comp( buffer, attr ):
    """Decode comp, binary values."""
    final, alpha, length, scale, precision, signed, dec_sign = attr.size_scale_precision
    if length <= 4:
        sc, bytes = ’>h’, 2
    elif length <= 9:
        sc, bytes = ’>i’, 4
    else:
        sc, bytes = ’>q’, 8
    n= struct.unpack( sc, buffer )
    return decimal.Decimal( n[0] )
```

Class-level logger

```
Character_File.log= logging.getLogger( Character_File.__qualname__ )
```

EBCDIC File The EBCDIC files require specific physical “Record Format” (RECFM) assistance. These classes define a number of Z/OS RECFM conversion. We recognize four actual RECFM’s plus an additional special case.

- F - Fixed.
- FB - Fixed Blocked.
- V - Variable, data must have the RDW word preserved.
- VB - Variable Blocked, data must have BDW and RDW words.
- N - Variable, but no BDW or RDW words. This involves some buffer management magic to recover the records properly.

Note: “IBM z/Architecture mainframes are all big-endian”.

class `cobol.`**`RECFM_Parser`**

This class hierarchy breaks up EBCDIC files into records.

```
class RECFM_Parser:
    """Parse a physical file format."""
    def record_iter( self ):
        """Return each physical record, stripped of headers."""
        raise NotImplementedError
    def used( self, bytes ):
        """The number of bytes actually consumed.
        Only really relevant for RECFM_N subclass to handle variable-length
        records with no RDW/BDW overheads.
        """
        pass
```

class `cobol.`**`RECFM_F`**

Simple fixed-length records. No header words.

```
class RECFM_F(RECFM_Parser):
    """Parse RECFM=F; the lrecl is the length of each record."""
    def __init__( self, source, lrecl=None ):
        """
        :param source: the file
        :param lrecl: the record length.
        """
        super().__init__()
        self.source= source
        self.lrecl= lrecl
    def record_iter( self ):
        data= self.source.read(self.lrecl)
        while len(data) != 0:
            yield data
            data= self.source.read(self.lrecl)
    def rdw_iter( self ):
        """Yield rows with RDW, effectively RECFM_V format."""
        for row in self.record_iter():
            yield struct.pack( ">H2x", len(row)+4 )+row
```

class cobol.**RECFM_FB**

Simple fixed-blocked records. No header words.

```
class RECFM_FB( RECFM_F ):
    """Parse RECFM=FB; the lrecl is the length of each record.

    It's not clear that there's any difference between F and FB.
    """
    pass
```

class cobol.**RECFM_V**

Variable-length records. Each record has an RDW header word with the length.

```
class RECFM_V(RECFM_Parser):
    """Parse RECFM=V; the lrecl is a maximum, which we ignore."""
    def __init__( self, source, lrecl=None ):
        """
        :param source: the file
        :param lrecl: a maximum, but it's ignored.
        """
        super().__init__()
        self.source= source
    def record_iter( self ):
        """Iterate over records, stripped of RDW's."""
        for rdw, row in self._data_iter():
            yield row
    def rdw_iter( self ):
        """Iterate over records which include the 4-byte RDW."""
        for rdw, row in self._data_iter():
            yield rdw+row
    def _data_iter( self ):
        rdw= self.source.read(4)
        while len(rdw) != 0:
            size = struct.unpack( ">H2x", rdw )[0]
            data= self.source.read( size-4 )
            yield rdw, data
            rdw= self.source.read(4)
```

We might want to implement the `RECFM_Parser.used()` method to compare the number of bytes used against the RDW size.

class cobol.**RECFM_VB**

Variable-length, blocked records. Each block has a BDW; each record has an RDW header word. These BDW and RDW describe the structure of the file.

```
class RECFM_VB(RECFM_Parser):
    """Parse RECFM=VB; the lrecl is a maximum, which we ignore."""
    def __init__( self, source, lrecl=None ):
        """
        :param source: the file
        :param lrecl: a maximum, but it’s ignored.
        """
        super().__init__()
        self.source= source
    def record_iter( self ):
        """Iterate over records, stripped of RDW’s."""
        for rdw, row in self._data_iter():
            yield row
    def rdw_iter( self ):
        """Iterate over records which include the 4-byte RDW."""
        for rdw, row in self._data_iter():
            yield rdw+row
    def bdw_iter( self ):
        """Iterate over blocks, which include 4-byte BDW and records with 4-byte RDW’s."""
        bdw= self.source.read(4)
        while len(bdw) != 0:
            blksize = struct.unpack( ">H2x", bdw )[0]
            block_data= self.source.read( blksize-4 )
            yield bdw+data
            bdw= self.source.read(4)
    def _data_iter( self ):
        bdw= self.source.read(4)
        while len(bdw) != 0:
            blksize = struct.unpack( ">H2x", bdw )[0]
            block_data= self.source.read( blksize-4 )
            offset= 0
            while offset != len(block_data):
                assert offset+4 < len(block_data), "Corrupted Data Block {!r}".format(block_data)
                rdw= block_data[offset:offset+4]
                size= struct.unpack( ">H2x", rdw )[0]
                yield rdw, block_data[offset+4:offset+size]
                offset += size
            bdw= self.source.read(4)
```

We might want to implement a generic `RECFM_Parser.used()` method to compare the number of bytes used against the RDW size and raise an exception in the event of a mismatch.

class `cobol.`**RECFM_N**

Variable-length records without RDW’s. Exasperating because we have to feed bytes to the buffer as needed until the record is complete.

```
class RECFM_N:
    """Parse RECFM=V without RDW (or RECFM=VB without BDW or RDW).
    The lrecl is ignored.
    """
    def __init__( self, source, lrecl=None ):
        """
        :param source: the file
        :param lrecl: a maximum, but it’s ignored.
        """
        super().__init__()
        self.source= source
        self.buffer= self.source.read( 32768 )
    def record_iter( self ):
        while len(self.buffer) != 0:
            yield self.buffer
            # What if used() is not called? This will loop forever!
    def used( self, bytes ):
        #print( "Consumed {0} Bytes".format(bytes) )
        self.buffer= self.buffer[bytes:]+self.source.read(32768-bytes)
```

class `cobol.`**`EBCDIC_File`**

This subclass handles EBCDIC conversion and COMP-3 packed decimal numbers. For this to work, the schema needs to use slightly different Cell-type conversions.

Otherwise, this is similar to processing simple character data.

```
class EBCDIC_File( Character_File ):
    """A COBOL "workbook" file with decoding functions for
    EBCDIC data. If a file_object is provided, it must be
    opened in byte mode, and no decoder can be used.
    """
    decoder= codecs.getdecoder('cp037')
    def __init__( self, name, file_object=None, schema=None, RECFM="N" ):
        """Prepare the workbook for reading.
        :param name: File name
        :param file_object: Optional file-like object.  If omitted, the named file is opened.
            The object must be opened in byte mode; no decoder should be used.
        :param schema: The schema to use.
        :param RECFM: The legacy Z/OS RECFM to use. This must be one
            of "F", "FB", "V", "VB". This is translated to an appropriate
            RECFM class: RECFM_F, RECFM_FB, RECFM_V, or RECFM_VB.
        """
        super().__init__( name, file_object, schema )
        if self.file_obj:
            self.the_file= None
            self.wb= self.file_obj
        else:
            self.the_file = open( name, 'rb' )
            self.wb= self.the_file
        self.schema= schema
        parser_class= {
            "F" : RECFM_F,
            "FB": RECFM_FB,
            "V" : RECFM_V,
            "VB": RECFM_VB,
            "N":  RECFM_N,
            }[RECFM]
        self.parser= parser_class(self.wb, schema.lrecl())
```

`EBCDIC_File.`**`rows_of`**`(`*sheet*`)`

We must extend the `workbook.Character_File.rows_of()` method to deal with two issues:

- If the schema depends on a variably located DDE, then we need to do the `cobol.defs.setSizeAndOffset()` function using the DDE. This is done automagically by the `ODO_LazyRow` object.
- The legacy Z/OS RECFM details.
  - We might have F or FB files, which are simply long runs of EBCDIC bytes with no line breaks. The LRECL must match the DDE.

  - We might have V (or VB) which have 4-byte header on each row (plus a 4-byte header on each block.) The LRECL doesn't matter.
  - We can tolerate the awful situation where it's variable length (Occurs Depending On) but there are no RECFM=V or RECFM=VB header words. We call this RECFM=N. We fetch an oversized buffer and push back bytes beyond the end of the record.

This means that the `super().rows_of( sheet )` has been replaced with a RECFM-aware byte-parser. This byte parser may involve a back-and-forth to handle RECFM=N. In the case of RECFM=N, we provide an overly-large buffer (32768 bytes) and after any size and offset calculations, the `row._size` shows how many bytes were actually used.

```
def rows_of( self, sheet ):
    """Iterate through all "rows" of this "sheet".
    Really, this means all records of this COBOL file.

    Note the handshake with RECFM parser to show how many
    bytes were really needed.  For RECFM_N, this is important.
    For other RECFM, this is ignored.

    :py:class:`ODO_LazyRow` may adjust the schema
    if it has an Occurs Depending On.
    """
    for data in self.parser.record_iter():
        row= ODO_LazyRow( sheet, data=data )
        self.parser.used(sheet.schema.lrecl())
        yield row
```

The following functions are used to do data conversions for COBOL EBCDIC files. Text requires using a codec to translate EBCDIC-encoded characters.

```
@staticmethod
def text( buffer, attr ):
    """Extract a text field's value."""
    text, size = EBCDIC_File.decoder(buffer)
    return text

@staticmethod
def number_display( buffer, attr ):
    """Extract a numeric field's value."""
    text, size = EBCDIC_File.decoder(buffer)
    return Character_File.number_display( text, attr )
```

ASCII File We could define a subclass for files encoded in ASCII which contain COMP and COMP-3 values.

This is left as a future extension.

COBOL Loader Module – Parse COBOL Source to Load a Schema

Parsing a spreadsheet schema is relatively easy: see *Schema Loader Module – Load Embedded or External Schema*. Parsing a COBOL schema, however, is a bit more complex.

Two of the three problems associated with COBOL are solved in *The COBOL Package*. What remains is parsing the COBOL source to extract a schema.

A new `schema.loader.ExternalSchemaLoader` subclass is required to parse the DDE sublanguage of COBOL. This loader will build the hierarchical DDE from the COBOL source, decorate this with size and offset information, then flatten it into a simple `schema.Schema` instance.

Load A Schema Use Case

The goal of use case is to load a schema encoded in COBOL. This breaks down into two steps.

1. Parse the COBOL "copybook" source file.
2. Produce a `schema.Schema` instance that can be used to access data in a COBOL file.

Ideally, this will look something like the following.

```
with open("sample/zipcty.cob", "r") as cobol:
    schema= stingray.cobol.loader.COBOLSchemaLoader( cobol ).load()
    #pprint.pprint( schema )
for filename in 'sample/zipcty1', 'sample/zipcty2':
    with stingray.cobol.Character_File( filename, schema=schema ) as wb:
        sheet= wb.sheet( filename )
        counts= process_sheet( sheet )
        pprint.pprint( counts )
```

Step 1 is to open the COBOL DDE "copybook" file, `zipcty.cob` that defines the layout. We build the schema using a `cobol.loader.COBOLSchemaLoader`.

Step 2 is to open the source data, `zipcty1` with the data. We've made a `sheet.Sheet` from the file: the sheet's name is `"zipcty1"`, the the schema is the external provided when we opened the `cobol.Character_File`.

Once the sheet is available, we can then run some function, `process_sheet`, on the sheet. This will use the `sheet.Sheet` API to process rows and cells of the sheet. Each piece of source data is loaded as a kind of `cell.Cell`.

We can then use appropriate conversions to recover Python objects. This leads us to the second use case.

Here's what a `process_sheet()` function might look like in this context.

```
def process_sheet( sheet ):
    schema_dict= dict( (a.name, a) for a in sheet.schema )
    schema_dict.update( dict( (a.path, a) for a in sheet.schema ) )

    counts= { 'read': 0 }

    row_iter= sheet.rows()
    header= header_builder( next(row_iter), schema_dict )
    print( header )
    for row in row_iter:
        detail= row_builder( row, schema_dict )
        print( detail )
        counts['read'] += 1
    return counts
```

First, we've build two versions of the schema, indexed by low-level item name and the full path to an item. In some cases, the low-level DDE items are unique, and the paths are not required. In other cases, the paths are required.

We've initialized some record counts, always a good practice.

We've fetched the first record and used some function named `header_builder()` to transform the record into a header, which we print.

We've fetched all other records and used a function named `row_builder()` to transform every following record into details, which we also print.

This shows a physical head-tail processing. In some cases, there's an attribute which differentiates headers, body and trailers.

Use A Schema Use Case

The goal of this use case is to build usable Python objects from the source file data.

For each row, there’s a two-step operation.

1. Access elements of each row using the COBOL DDE structure.
2. Build Python objects from the Cells found in the row.

Generally, we must use lazy evaluation as shown in this example:

```
def header_builder(row, schema):
    return dict(
        file_version_year= row.cell(schema[’FILE-VERSION-YEAR’]).to_str(),
        file_version_month= row.cell(schema[’FILE-VERSION-MONTH’]).to_str(),
        copyright_symbol= row.cell(schema[’COPYRIGHT-SYMBOL’]).to_str(),
        tape_sequence_no= row.cell(schema[’TAPE-SEQUENCE-NO’]).to_str(),
    )

def row_builder(row, schema):
    return dict(
        zip_code= row.cell(schema[’ZIP-CODE’]).to_str(),
        update_key_no= row.cell(schema[’UPDATE-KEY-NO’]).to_str(),
        low_sector= row.cell(schema[’COUNTY-CROSS-REFERENCE-RECORD.ZIP-ADD-ON-RANGE.ZIP-ADD-ON-LOW-NO
        low_segment= row.cell(schema[’COUNTY-CROSS-REFERENCE-RECORD.ZIP-ADD-ON-RANGE.ZIP-ADD-ON-LOW-N
        high_sector= row.cell(schema[’COUNTY-CROSS-REFERENCE-RECORD.ZIP-ADD-ON-RANGE.ZIP-ADD-ON-HIGH-
        high_segment= row.cell(schema[’COUNTY-CROSS-REFERENCE-RECORD.ZIP-ADD-ON-RANGE.ZIP-ADD-ON-HIGH
        state_abbrev= row.cell(schema[’STATE-ABBREV’]).to_str(),
        county_no= row.cell(schema[’COUNTY-NO’]).to_str(),
        county_name= row.cell(schema[’COUNTY-NAME’]).to_str(),
    )
```

Each cell is accessed in a three-step operation.

1. Get the schema information via `schema[’shortname’]` or `schema[’full.path.name’]`
2. Build the `Cell` using the schema information via `row.cell(...)`.
3. Convert the `Cell` to our target type via `...to_str()`.

We **must** do this in steps because the COBOL records may have invalid fields, or `REDEFINES` or `OCCURS DEPENDING ON` clauses.

If we want to build higher-level, pure Python objects associated with some application, we’ll do this.

```
def build_object(row, schema):
    return Object( **row_builder(row, schema) )
```

We’ll simply assure that the row’s dictionary keys are the proper keyword arguments for our application class definitions.

When we have indexing to do, this is only slightly more complex. The resulting object will be a list-of-list structure, and we apply the indexes in the order from the original DDE definition to pick apart the lists.

Extensions and Special Cases

The typical use cases is something like the following:

```
with open("sample/zipcty.cob", "r") as cobol:
    schema= stingray.cobol.loader.COBOLSchemaLoader( cobol ).load()
with stingray.cobol.Character_File( filename, schema=schema ) as wb:
    sheet= wb.sheet( filename )
    for row in sheet.rows():
        dump( schema, row )
```

This will use the default parsing to create a schema from a DDA and process a file, dumping each record.

There are two common extension:

- new lexical scanner, and
- different ODO handling.

To change lexical scanners, we create a new subclass of the parser.

We use this by subclassing `cobol.COBOLSchemaLoader`.

```
class MySchemaLoader( cobol.COBOLSchemaLoader ):
    lexer_class= cobol.loader.Lexer_Long_Lines
```

This will use a different lexical scanner when parsing a DDE file.

We may also need to change the record factory. This involves two separate extensions. We must extend the `cobol.loader.RecordFactory` to change the features. Then we can extend `cobol.loader.COBOLSchemaLoader` to use this record factory.

```
class ExtendedRecordFactory( cobol.loader.RecordFactory ):
    occurs_dependingon_class= stingray.cobol.defs.OccursDependingOnLimit
    #Default is occurs_dependingon_class= stingray.cobol.defs.OccursDependingOn

class MySchemaLoader( cobol.loader.COBOLSchemaLoader ):
    record_factory_class= ExtendedRecordFactory
```

This will use a different record factory to elaborate the details of the DDE.

Design

A DDE contains a recursive definition of a COBOL group-level DDE. There are two basic species of COBOL DDE's: elemetary items, which have a `PICTURE` clause, and group-level items, which contain lower-level items. There are several optional features of every DDE, including an `OCCURS` clause and a `REDEFINES` clause. In addition to the required picture clause, elementary items have an optional `USAGE` clause, and optional `SIGN` clause.

The `PICTURE` clause specifies how to interpret a sequence of bytes. The picture clause interacts with the optional `USAGE` clause, `SIGN` clause and `SYNCHRONIZED` clause to fully define the encoding. The picture clause uses a complex format of code characters to define either individual character bytes (when the usage is display) or pairs of decimal digit bytes (when the usage is `COMP-3`).

The `OCCURS` clause specifies an array of elements. If the occurs clause appears on a group level item, the sub-record is repeated. If the occurs clause appears on an elementary item, that item is repeated.

An **occurs depending on** (ODO) makes the positions of each field dependent on actual data present in the record. This is a rare, but necessary complication.

The `REDEFINES` clause defines an alias for input bytes. When some field *R* redefines a previously defined field *F*, the storage bytes are used for both *R* and *F*. The record structure itself does not provide a way to disambiguate the interpretation of the bytes. Program logic must be examined to determine the conditions under which each interpretation is valid. It's entirely possible either interpretation has invalid fields.

DDE Class The parent class, `DDE`, defines the features of a group-level item. It supports the occurs and redefines features. It can contain a number of DDE items. The leaves of the tree define the features of an elementary item.

We could have a class hierarchy with group and elementary subclasses. The group level item could have a container for lower level items. The elementary class definition *could* add support for the picture clause, but remove the container for lower-level items.

On balance, it seems simpler to have one generic DDE node class and use optional fields than to create a proper subclass. There isn't a *good* reason for this. An **if** statement to look for an optional picture-clause is fairly rare.

The various optional clauses are handled using a variety of design patterns. The usage information, for instance, is used to create a **Strategy** object that is used to extract a field from a record's bytes.

The redefines information is used to create a **Strategy** object that computes the offset to a field. There are two variant strategies: locate the basis field and use that field's offset or use the end of the previous element as the offset.

This is further compounded by the Occurs Depending On (ODO) calculation which cannot be done statically or eagerly, but must be done dynamically and lazily based on live data.

DDE Post-processing We have a number of functions to traverse a DDE structure to write reports on the structure. The DDE has an `__iter__()` method which provides a complete pre-order depth-first traversal of the record structure.

Here are some functions which traverse the entire DDE structure.

- `cobol.defs.report()` reports on the DDE structure.
- `cobol.defs.source()` shows canonical source.
- `cobol.defs.search()` locates a name in DDE structure.
- `cobol.defs.resolver()` does name resolution throughout the DDE structure.
- `cobol.defs.setDimensionality()` walks up the hierarchy from each node to compute the net occurrences based on all parent OCCURS clauses.

Once there is data available, we have these additional functions.

- `cobol.defs.setSizeAndOffset()` computes the offset and size of each element.
- `cobol.dump()` dumps a record showing the original DDE and the values.

Note that `cobol.defs.setSizeAndOffset()` is recursive, not iterative. It needs to manage subtotals based on ascent and descent in the hierarchy.

DDE Parser A `cobol.loader.RecordFactory` object reads a file of text and either creates a DDE or raises an exception. If the text is a valid COBOL record definition, a DDE is created. If there are syntax errors, an exception is raised.

The `cobol.loader.RecordFactory` depends on a `cobol.loader.Lexer` instance to do lexical scanning of COBOL source. The lexical scanner can be subclassed to pre-process COBOL source. This is necessary because of the variety of source formats that are permitted. Shop standards may include or exclude features like program identification, line numbers, format control and other decoration of the input.

The `cobol.loader.RecordFactory.makeRecord()` method does the parsing of the record definition. Each individual DDE statement is parsed. The level number information is used to define the correct grouping of elements. When the structure(s) is parsed, it is decorated with size and offset information for each element.

Note that multiple 01 levels are possible in a single COBOL copybook. This is confusing and potentially complicated, but it occurs IRL.

Field Values The COBOL language, and IBM's extensions, provide for a number of usage options. In this application, three basic types of usage strategies are supported:

- **DISPLAY**. These are bytes, one per character, described by the picture clause. They can be EBCDIC or ASCII. We use the `codecs` module to convert EBCDIC characters to Unicode for further processing.
- **COMP**. These are binary fields of 2, 4 or 8 bytes, with the size implied by the picture clause.
- **COMP-3**. These are packed decimal fields, with the size derived from the picture clause; there are two digits packed into each byte, with an extra half-byte for a sign.

These require different strategies for decoding the input bytes.

Additional types include COMP-1 and COMP-2 which are single- and double-precision floating-point. They're rare enough that we ignore them.

Additional Requirements Support for Occurs Depending On is based several features of COBOL.

The syntax for ODO is more complex: `OCCURS [int TO] int [TIMES] DEPENDING [ON] name`. Compare this with simple `OCCURS int [TIMES]`.

This leads to variable byte positions for data items which follow the occurs clause, based on the *name* value.

This means that the offset is not necessarily fixed when there's a complex ODO. We'll have to make offset (and size) a property that has one of two strategies.

- Statically Located. The base case where offsets are static.
- Variably Located. The complex ODO situation where there's an ODO in the record. **All** ODO "depends on" fields become part of the offset calculation. This means we need an index for depends on clauses.

The technical buzzphrase is "a data item following, but not subordinate to, a variable-length table in the same level-01 record."

See http://publib.boulder.ibm.com/infocenter/comphelp/v7v91/index.jsp?topic=%2Fcom.ibm.aix.cbl.doc%2Ftptbl27.htm

These are the "Appendix D, Complex ODO" rules.

The design consequences are these.

1. There are three species of relationships between DDE elements: Predecessor/Successor, Parent/Child (or Group/Elementary), and Redefines. Currently, the pred/succ relationship is implied by the parent having a sequence of children. We can't easily find a predecessor without a horrible $\mathbf{O}(n)$ search.
2. There are two strategies for doing offset/size calculations.
   - Statically Located. The `cobol.defs.setSizeAndOffset()` function can be used once, right after the schema is parsed.
   - Variably Located. The calculation of size and offset is based on live data. The `cobol.defs.setSizeAndOffset()` function must be used after the row is fetched but before any other processing.

     This is done automagically by a `sheet.LazyRow` object.

The offset calculation can be seen as a recursive trip "up" the tree following redefines, predecessor and parent relationships (in that order) to calculate the size of everything prior to the element in question. We could make offset and total size into properties which do this recursive calculation.

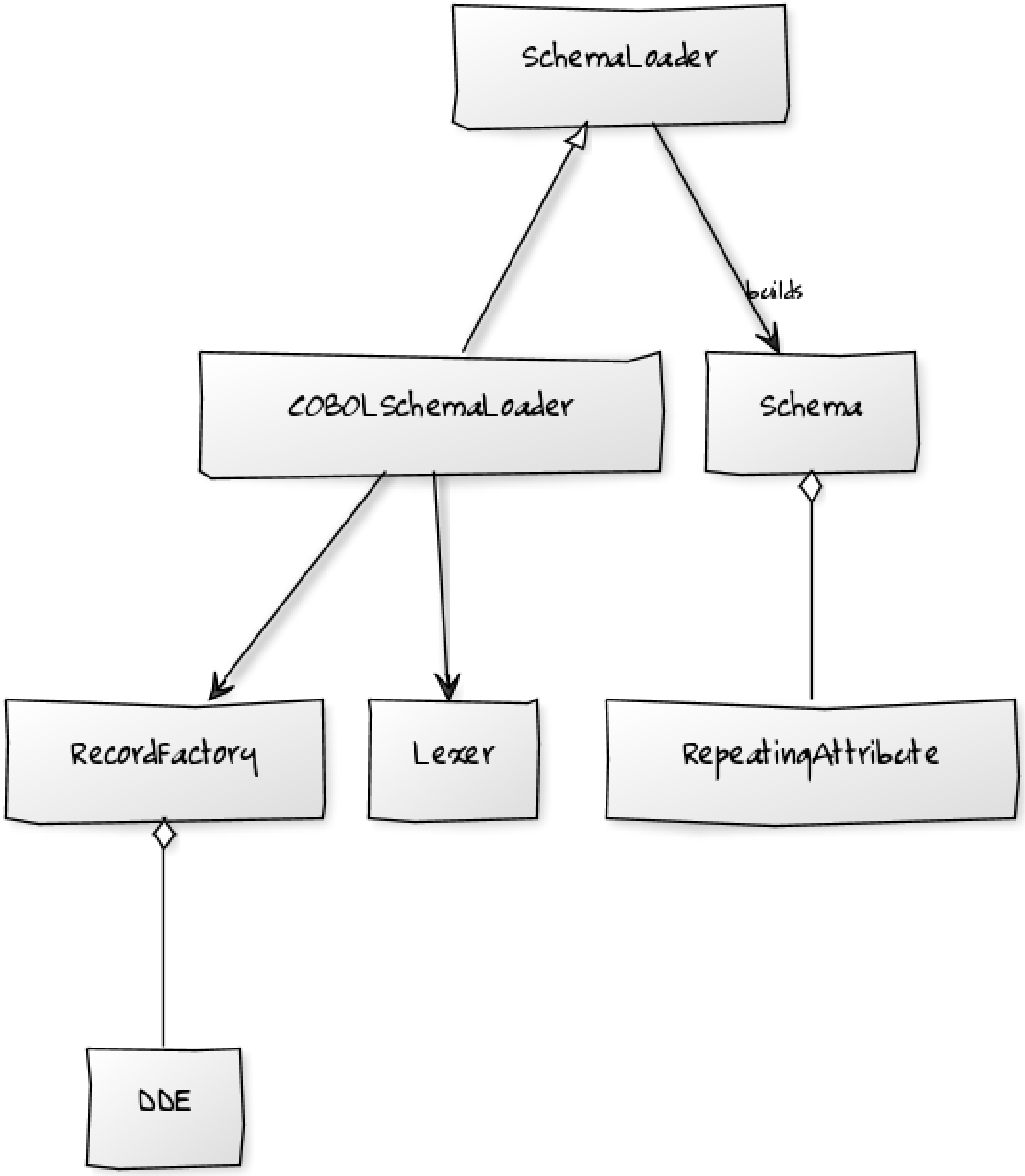
The "size" of a elementary items is still simply based on the picture. For group items, however, size becomes based on total size which in turn, may be based on ODO data.

Todo

88-level items could create boolean-valued properties.

Model

```
http://yuml.me/diagram/scruffy;/class/
#cobol_loader,
[Schema]<>-[RepeatingAttribute],
[SchemaLoader]-builds->[Schema],
[SchemaLoader]^[COBOLSchemaLoader],
[COBOLSchemaLoader]->[Lexer],
[COBOLSchemaLoader]->[RecordFactory],
[RecordFactory]<>-[DDE].
```



Overheads

Ultimately, we're writing a new `schema.loader.ExternalSchemaLoader`. The purpose of this is to build a `schema.Schema` instance from COBOL source instead of some other source.

```
"""stingray.cobol.loader -- Parse a COBOL DDE and build a usable Schema."""
import re
from collections import namedtuple, Iterator
import logging
import weakref
import warnings

import stingray.schema.loader
import stingray.cobol
import stingray.cobol.defs
```

A module-level logger.

```
logger= logging.getLogger( __name__ )
```

Parsing Exceptions

class `cobol.loader.`**`SyntaxError`**

These are compilation problems. We have syntax which is utterly baffling.

```
class SyntaxError( Exception ):
    """COBOL syntax error."""
    pass
```

Picture Clause Parsing

Picture clause parsing is done as the DDE element is created. Not for a great reason. It's derived data from the source picture clause.

It could be done in the parser, also.

class `cobol.loader.`**`Picture`**

Final the final picture

Alpha boolean; True if any `"X"` or `"A"`; False if all `"9"` and related

Length length of the final picture

Scale count of `"P"` positions, often zero

Precision digits to the right of the decimal point

Signed boolean; True if any `"S"`, `"-"` or related

Decimal `"."` or `"V"` or `None`

```
Picture = namedtuple( 'Picture',
    'final, alpha, length, scale, precision, signed, decimal' )
```

`cobol.loader.`**`picture_parser`**`(`*pic*`)`

```
def picture_parser( pic ):
    """Rewrite a picture clause to eliminate ()’s, S’s, V’s, P’s, etc.
    :param pic: Sounce text.
    :returns: Picture instance.
    """
    out= []
    scale, precision, signed, decimal = 0, 0, False, None
    char_iter= iter(pic)
    for c in char_iter:
        if c in (’A’,’B’,’X’,’Z’,’9’,’0’,’/’,’,’,’+’,’-’,’*’,’$’):
            out.append( c )
            if decimal: precision += 1
        elif c == ’D’:
            nc= next(char_iter)
            assert nc == "B", "picture error in {0!r}".format(pic)
            out.append( "DB" )
            signed= True
        elif c ==  ’C’:
            nc= next(char_iter)
            assert nc == "R", "picture error in {0!r}".format(pic)
            out.append( "CR" )
            signed= True
        elif c == ’(’:
            irpt= 0
            try:
                for c in char_iter:
                    if c == ’)’: break
                    irpt = 10*irpt + int( c )
            except ValueError as e:
                raise SyntaxError( "picture error in {0!r}".format(pic) )
            assert c == ’)’,  "picture error in {0!r}".format(pic)
            out.append( (irpt-1)*out[-1] )
        elif c == ’S’:
            # silently drop an "S".
            # Note that ’S’ plus a SIGN SEPARATE option increases the size of the picture!
            signed= True
        elif c  == ’P’:
            # "P" sets scale and isn’t represented.
            scale += 1
        elif c  == "V":
            # "V" sets precision and isn’t represented.
            decimal= "V"
        elif c  == ".":
            decimal= "."
            out.append( "." )
        else:
            raise SyntaxError( "Picture error in {!r}".format(pic) )

    final= "".join( out )
    alpha= (’A’ in final) or (’X’ in final) or (’/’ in final)
    logger.debug( "PIC {0} {1} {2} {3} {4}".format(pic, final, alpha, scale, precision) )
    # Note: Actual bytes consumed depends on len(final) and usage!
    return Picture( final, alpha, len(final), scale,
        precision, signed, decimal)
```

Lexical Scanning

The lexical scanner can be subclassed to extend its capability. The default lexical scanner provides a `Lexer.clean()` method that simply removes comments. This may need to be overridden to remove line numbers (from positions 72-80), module identification (from positions 1-5), and format control directives.

class `cobol.loader.`**`Lexer`**

Basic lexer that simply removes comments and the first six positions of each line.

```
class Lexer:
    """Lexical scanner for COBOL.  Iterates over tokens in source text."""
    separator= re.compile( r'[.,;]?\s' )
    quote1= re.compile( r"'[^']*'" )
    quote2= re.compile( r'"[^"]*"' )
    def __init__( self, replacing=None ):
        self.log= logging.getLogger( self.__class__.__qualname__ )
        self.replacing= replacing or []
```

`Lexer.`**`clean`**`(`*line*`)`

```
def clean( self, line ):
    """Default cleaner removes positions 0:6."""
    return line[6:].rstrip()
```

`Lexer.`**`scan`**`(`*text*`)`

```
def scan( self, text ):
    """Locate the next token in the input stream.
    - Clean 6-char lead-in plus trailing whitespace
    - Add one extra space to distinguish end-of-line ``'. '``
      from picture clause.
    """
    if isinstance(text, (str, bytes)):
        text= text.splitlines()
    self.all_lines= ( self.clean(line) + ' '
        for line in text )
    # Remove comments and blank lines
    self.lines = ( line for line in self.all_lines
        if line and line[0] not in ('*', '/') )
    for line in self.lines:
        logger.debug( line )
        if len(line) == 0: continue
        for old, new in self.replacing:
            line= line.replace(old,new)
        if self.replacing: logger.debug( line )
        current= line.lstrip()
        while current:
            if current[0] == "'":
                # apostrophe string, break on balancing apostrophe
                match= self.quote1.match( current )
                space= match.end()
            elif current[0] == '"':
                # quote string, break on balancing quote
                match= self.quote2.match( current )
                space= match.end()
            else:
                match= self.separator.search( current )
                space= match.start()
                if space == 0: # starts with separator
                    space= match.end()-1
            token, current = current[:space], current[space:].lstrip()
            self.log.debug( token )
            yield token
```

class `cobol.loader.`**`Lexer_Long_Lines`**

More sophisticated lexer that removes the first six positions of each line. If the line is over 72 positions, it also removes positions [71:80]. Since it's an extension to `cobol.loader.Lexer`, it also removes comments.

```
class Lexer_Long_Lines( Lexer ):

    def clean( self, line ):
        """Remove positions 72:80 and 0:6."""
        if len(line) > 72:
            return line[6:72].strip()
        return line[6:].rstrip()
```

We use this by subclassing `cobol.COBOLSchemaLoader`.

```
class MySchemaLoader( cobol.COBOLSchemaLoader ):
    lexer_class= cobol.Lexer_Long_Lines
```

Parsing

The `cobol.loader.RecordFactory` class is the parser for record definitions. The parser has three basic sets of methods:

1. clause parsing methods,
2. element parsing methods and
3. complete record layout parsing.

Parsing a record layout involves parsing a sequence of elements and assembling them into a proper structure. Each element consists of a sequence of individual clauses.

The picture clauses are parsed separately by the DDE during its initialization.

class `cobol.loader.`**`RecordFactory`**

```
class RecordFactory:
    """Parse a copybook, creating a DDE structure."""
    noisewords= {"WHEN","IS","TIMES"}
    keywords= {"BLANK","ZERO","ZEROS","ZEROES","SPACES",
        "DATE","FORMAT","EXTERNAL","GLOBAL",
        "JUST","JUSTIFIED","LEFT","RIGHT"
        "OCCURS","DEPENDING","ON","TIMES",
        "PIC","PICTURE",
        "REDEFINES","RENAMES",
        "SIGN","LEADING","TRAILING","SEPARATE","CHARACTER",
        "SYNCH","SYNCHRONIZED",
        "USAGE","DISPLAY","COMP-3",
        "VALUE","."}

    redefines_class= stingray.cobol.defs.Redefines
    successor_class= stingray.cobol.defs.Successor
    group_class= stingray.cobol.defs.Group
    display_class= stingray.cobol.defs.UsageDisplay
    comp_class= stingray.cobol.defs.UsageComp
    comp3_class= stingray.cobol.defs.UsageComp3
    occurs_class= stingray.cobol.defs.Occurs
    occurs_fixed_class= stingray.cobol.defs.OccursFixed
    occurs_dependingon_class= stingray.cobol.defs.OccursDependingOn

    def __init__( self ):
        self.lex= None
        self.token= None
        self.context= []
        self.log= logging.getLogger( self.__class__.__qualname__ )
```

Each of these parsing functions has a precondition of the last examined token in `self.token`. They have a post-condition of leaving a **not**-examined token in `self.token`.

```
def picture( self ):
    """Parse a PICTURE clause."""
    self.token= next(self.lex)
    if self.token == "IS":
        self.token= next(self.lex)
    pic= self.token
    self.token= next(self.lex)
    return pic

def blankWhenZero( self ):
    """Gracefully skip over a BLANK WHEN ZERO clause."""
    self.token= next(self.lex)
    if self.token == "WHEN":
        self.token= next(self.lex)
    if self.token in {"ZERO","ZEROES","ZEROS"}:
        self.token= next(self.lex)

def justified( self ):
    """Gracefully skip over a JUSTIFIED clause."""
    self.token= next(self.lex)
    if self.token == "RIGHT":
        self.token= next(self.lex)

def occurs( self ):
    """Parse an OCCURS clause."""
    occurs= next(self.lex)
    if occurs == "TO":
        # format 2: occurs depending on with assumed 1 for the lower limit
        return self.occurs2( '' )
    self.token= next(self.lex)
    if self.token == "TO":
        # format 2: occurs depending on
        return self.occurs2( occurs )
    else:
        # format 1: fixed-length
        if self.token == "TIMES":
            self.token= next(self.lex)
        self.occurs_cruft()
        return self.occurs_fixed_class(occurs)

def occurs_cruft( self ):
    """Soak up additional key and index sub-clauses."""
    if self.token in {"ASCENDING","DESCENDING"}:
        self.token= next(self.lex)
    if self.token == "KEY":
        self.token= next(self.lex)
    if self.token == "IS":
        self.token= next(self.lex)
    # get key data names
    while self.token not in self.keywords:
        self.token= next(self.lex)
    if self.token == "INDEXED":
        self.token= next(self.lex)
    if self.token == "BY":
        self.token= next(self.lex)
    # get indexed data names
    while self.token not in self.keywords:
        self.token= next(self.lex)

def occurs2( self, lower ):
    """Parse the [Occurs n TO] m Times Depending On name"""
    self.token= next(self.lex)
    upper= self.token # May be significant as a default size.
    default_size= int(upper)
    self.token= next(self.lex)
    if self.token == "TIMES":
        self.token= next(self.lex)
    if self.token == "DEPENDING":
        self.token= next(self.lex)
    if self.token == "ON":
        self.token= next(self.lex)
    name= self.token
    self.token= next(self.lex)
    self.occurs_cruft()

    return self.occurs_dependingon_class( name, default_size )
    #raise stingray.cobol.defs.UnsupportedError( "Occurs depending on" )

def redefines( self ):
    """Parse a REDEFINES clause."""
    redef= next(self.lex)
    self.token= next(self.lex)
    return self.redefines_class(name=redef)
```

A RENAMES creates an alternative group-level name for some elementary items. It's considered bad practice.

```
def renames( self ):
    """Raise an exception on a RENAMES clause."""
    ren1= next(self.lex)
    self.token= next(self.lex)
    if self.token in {"THRU","THROUGH"}:
        ren2= next(self.lext)
        self.token= next(self.lex)
    raise stingray.cobol.defs.UnsupportedError( "Renames clause" )
```

There are two variations on the SIGN clause syntax.

```
def sign1( self ):
    """Raise an exception on a SIGN clause."""
    self.token= next(self.lex)
    if self.token == "IS":
        self.token= next(self.lex)
    if self.token in {"LEADING","TRAILING"}:
        self.sign2()
    # TODO: this may change the size to add a sign byte
    raise stingray.cobol.defs.UnsupportedError( "Sign clause" )
def sign2( self ):
    """Raise an exception on a SIGN clause."""
    self.token= next(self.lex)
    if self.token == "SEPARATE":
        self.token= next(self.lex)
    if self.token == "CHARACTER":
        self.token= next(self.lex)
    raise stingray.cobol.defs.UnsupportedError( "Sign clause" )

def synchronized( self ):
    """Raise an exception on a SYNCHRONIZED clause."""
    self.token= next(self.lex)
    if self.token == "LEFT":
        self.token= next(self.lex)
    if self.token == "RIGHT":
        self.token= next(self.lex)
    raise stingray.cobol.defs.UnsupportedError( "Synchronized clause" )
```

There are two variations on the USAGE clause syntax.

```
def usage( self ):
    """Parse a USAGE clause."""
    self.token= next(self.lex)
    if self.token == "IS":
        self.token= next(self.lex)
    use= self.token
    self.token= next(self.lex)
    return self.usage2( use )
def usage2( self, use ):
    """Create a correct Usage instance based on the USAGE clause."""
    if use == "DISPLAY": return self.display_class(use)
    elif use == "COMPUTATIONAL": return self.comp_class(use)
    elif use == "COMP": return self.comp_class(use)
    elif use == "COMPUTATIONAL-3": return self.comp3_class(use)
    elif use == "COMP-3": return self.comp3_class(use)
    else: raise SyntaxError( "Unknown usage clause {!r}".format(use) )
```

For 88-level items, the value clause can be quite long. Otherwise, it's just a single item. We have to absorb all quoted literal values. It may be that we have to absorb all non-keyword values.

```
def value( self ):
    """Parse a VALUE clause."""
    if self.token == "IS":
        self.token= next(self.lex)
    lit= [next(self.lex),]
    self.token= next(self.lex)
    while self.token not in self.keywords:
        lit.append( self.token )
        self.token= next(self.lex)
    return lit
```

This fits the generator design pattern well. The low-level `RecordFactory.dde_iter()` method emits individual DDE statements. These will be assembled into an overall record definition, below.

RecordFactory.**dde_iter**(*lexer*)

```
def dde_iter( self, lexer ):
    """Create a single DDE from an entry of clauses."""
    self.lex= lexer

    for self.token in self.lex:
        # Start with the level.
        level= self.token

        # Pick off a name, if present
        self.token= next(self.lex)
        if self.token in self.keywords:
            name= "FILLER"
        else:
            name= self.token
            self.token= next(self.lex)

        # Defaults
        usage= self.display_class( "" )
        pic= None
        occurs= self.occurs_class()
        redefines= None # set to Redefines below or by addChild() to Group or Successor

        # Accumulate the relevant clauses, dropping noise words and irrelevant clauses.
        while self.token and self.token != ’.’:
            if self.token == "BLANK":
                self.blankWhenZero()
            elif self.token in {"EXTERNAL","GLOBAL"}:
                self.token= next(self.lex)
            elif self.token in {"JUST","JUSTIFIED"}:
                self.justified()
            elif self.token == "OCCURS":
                occurs= self.occurs()
            elif self.token in {"PIC","PICTURE"}:
                pic= self.picture()
            elif self.token == "REDEFINES":
                # Must be first and no other clauses allowed.
                # Special case: simpler if 01 level ignores this clause.
                clause= self.redefines()
                if level == ’01’:
                    self.log.info( "Ignoring top-level REDEFINES" )
                else:
                    redefines= clause
            elif self.token == "RENAMES":
                self.renames()
            elif self.token == "SIGN":
                self.sign1()
            elif self.token in {"LEADING","TRAILING"}:
                self.sign2()
            elif self.token == "SYNCHRONIZED":
                self.synchronized()
            elif self.token == "USAGE":
                usage= self.usage()
            elif self.token == "VALUE":
                self.value()
            else:
                try:
                    # Keyword USAGE is optional
                    usage= self.usage2( self.token )
                    self.token= next(self.lex)
                except SyntaxError as e:
                    raise SyntaxError( "{!r} unrecognized".format(self.token) )
        assert self.token == "."

        # Create and yield the DDE
        if pic:
            # Parse the picture; update the USAGE clause with details.
            sizeScalePrecision= picture_parser( pic )
            usage.setTypeInfo(sizeScalePrecision)

            # Build an elementary DDE
            dde= stingray.cobol.defs.DDE(
                level, name, usage=usage, occurs=occurs, redefines=redefines,
                pic=pic, sizeScalePrecision=sizeScalePrecision )
        else:
            # Build a group-level DDE
            dde= stingray.cobol.defs.DDE(
                level, name, usage=usage, occurs=occurs, redefines=redefines )

        yield dde
```

Note that some clauses (like `REDEFINES`) occupy a special place in COBOL syntax. We're not fastidious about enforcing COBOL semantic rules. Presumably the source is proper COBOL and was actually used to create the source file.

The overall parsing method, `RecordFactory.makeRecord()` is an iterator that yields the top-level parsed items. It uses the `RecordFactory.dde_iter()` to get tokens and accumulates a proper hierarchy of individual DDE instances.

This will yield the `01`-level records. Generally, there's only one.

`RecordFactory.`**`makeRecord`**`(`*lexer*`)`

```
def makeRecord( self, lexer ):
    """Parse an entire copybook block of text."""
    # Parse the first DDE and establish the context stack.
    ddeIter= self.dde_iter( lexer )
    top= next(ddeIter)
    top.top, top.parent = weakref.ref(top), None
    top.allocation= stingray.cobol.defs.Group()
    self.context= [top]
    for dde in ddeIter:
        #print( dde, ":", self.context[-1] )
        # If a lower level or same level, pop context
        while self.context and dde.level <= self.context[-1].level:
            self.context.pop()

        if len(self.context) == 0:
            # Special case of multiple 01 levels.
            self.log.info( "Multiple {0} levels".format(top.level) )
            self.decorate( top )
            yield top
            # Create a new top with this DDE.
            top= dde
            top.top, top.parent = weakref.ref(top), None
            top.allocation= stingray.cobol.defs.Group()
            self.context= [top]
        else:
            # General case.
            # Make this DDE part of the parent DDE at the top of the context stack
            self.context[-1].addChild( dde )
            # Push this DDE onto the context stack
            self.context.append( dde )
            # Handle special case of "88" level children.
            if dde.level == ’88’:
                assert dde.parent().picture, "88 not under elementary item"
                dde.size= dde.parent().size
                dde.usage= dde.parent().usage

    self.decorate( top )
    yield top
```

`RecordFactory.`**`decorate`**`(`*top*`)`

The final stages of compilation:

- Resolve `REDEFINES` names using `cobol.defs.resolver()`.
- Push dimensionality down to each elementary item using `cobol.defs.setDimensionality()`.
- Work out size and offset, if possible. Use using `cobol.defs.setSizeAndOffset()` This depends on the presence of Occurs Depending On. If we can’t compute size and offset, it must be computed as each row is read. This is done automagically by a `sheet.LazyRow` object.

  Should we emit a warning? It’s not usually a mystery that the DDE involves Occurs Depending On.

```
def decorate( self, top ):
    """Three post-processing steps: resolver, size and offset, dimensionality."""
    stingray.cobol.defs.resolver( top )
    stingray.cobol.defs.setDimensionality( top )
    if top.variably_located:
        # Cannot establish all offsets and total sizes.
        pass # Log a warning?
    else:
        stingray.cobol.defs.setSizeAndOffset( top )
```

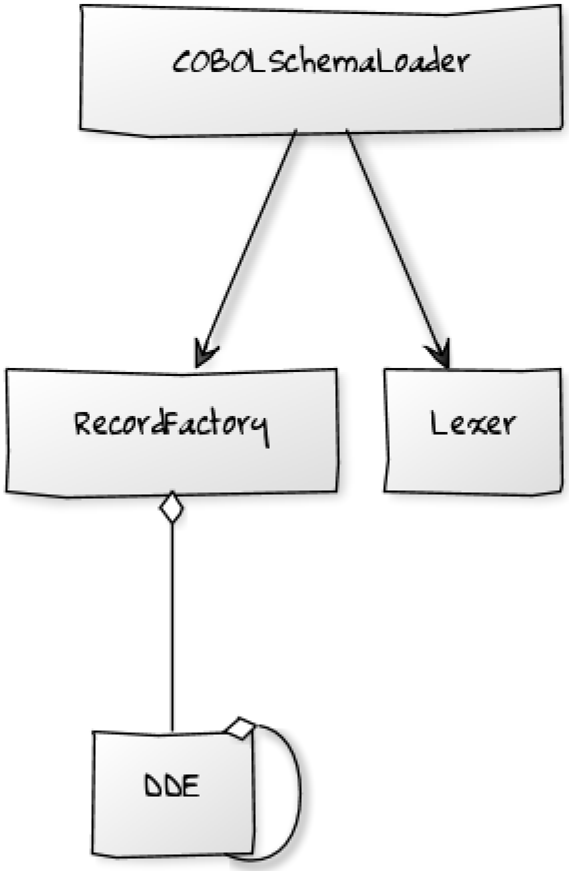
COBOL Schema Loader

Given a DDE, create a proper `schema.Schema` object which contains proper `schema.Attribute` objects for each group and elementary item in the DDE.

This schema, then, can be used with a COBOL workbook to fetch the rows and columns. Note that the conversions involved may be rather complex.

The `schema.Attribute` objects are built by a function that extracts relevant bits of goodness from a DDE.

```
http://yuml.me/diagram/scruffy;/class/
#cobol_loader_final,
[COBOLSchemaLoader]->[Lexer],
[COBOLSchemaLoader]->[RecordFactory],
[RecordFactory]<>-[DDE],
[DDE]<>-[DDE].
```



cobol.loader.**make_attr**()

```
def make_attr( aDDE ):
    attr= stingray.cobol.RepeatingAttribute(
        # Essential features:
        name= aDDE.name,
        size= aDDE.size,
        create= aDDE.usage.create_func,

        # COBOL extensions:
        dde= weakref.ref(aDDE),
    )
    aDDE.attribute= weakref.ref( attr )
    return attr
```

cobol.loader.**make_schema**()

The schema.Schema – as a whole – is built by a function that converts the DDE's into attributes.

This may need to be extended in case other DDE names (i.e. paths) are required in addition to the elementary names.

```
def make_schema( dde_iter ):
    schema= stingray.schema.Schema( dde=[] )
    for record in dde_iter:
        schema.info['dde'].append( record )
        for aDDE in record:
            attr= make_attr(aDDE)
            schema.append( attr )
    return schema
```

class `cobol.loader.`**`COBOLSchemaLoader`**

Here's the overall schema loader process: parse and then build a schema. This is consistent with the `schema.loader.ExternalSchemaLoader`. However, this is rarely precisely what we want. We're almost always going to break this down into separate steps.

```
class COBOLSchemaLoader( stingray.schema.loader.ExternalSchemaLoader ):
    """Parse a COBOL DDE and create a Schema.
    A subclass may define the lexer_class to customize
    parsing.
    """
    lexer_class= Lexer
    record_factory_class= RecordFactory
    def __init__( self, source, replacing=None ):
        self.source= source
        self.lexer= self.lexer_class( replacing )
        self.parser= self.record_factory_class()
```

`COBOLSchemaLoader.`**`load`**`()`

```
def load( self ):
    dde_iter= self.parser.makeRecord( self.lexer.scan(self.source) )
    schema= make_schema( dde_iter )
    return schema
```

The `replacing` keyword argument is a sequence of pairs: `[ ('old','new'), ...]`. The old text is replaced with the new text. This seems strange because it is. COBOL allows replacement text to permit reuse without name clashes.

Note that we provide the "replacing" option to the underlying Lexer. The lexical scanning includes any replacement text.

In some cases, we want to see the intermediate COBOL record definitions. In this case, we want to do something like the following function.

`cobol.loader.`**`COBOL_schema`**`(`*source*, *replacing=None*`)`

This function will parse the COBOL copybook, returning both the parsed COBOL as well as the final schema.

We may need to revise this function because we need a different lexer. We might have some awful formatting issue with the source that needs to be tweaked.

```
def COBOL_schema( source, replacing=None ):
    lexer= Lexer( replacing )
    parser= RecordFactory()
    dde_list= list( parser.makeRecord( lexer.scan(source) ) )
    schema= make_schema( dde_list )
    return dde_list, schema
```

COBOL Definitions Module – Handle COBOL DDE's

This is a small set of class definitions and functions that are used by `cobol.loader` as well as `cobol`.

The intent of this module is to avoid a few circular import dependencies.

The Architecture Problem

We have an issue of separation of three concerns:

- The underlying workbook and the parsing of CSV or XML or EBCDIC. This is the Physical Format.
- The logical layout or schema we're imposing on the workbook's data.
- The process of loading a schema, possibly using a meta-workbook. This includes the translation of COBOL notation into a useful schema.

Except for COBOL, a schema depends on a meta-workbook via a schema loader. But this is the limit of the relationship. We could say

$$S = L(w)$$

Or `schema= loader(workbook)`. This may involve a separate workbook file, a separate sheet within a file or even just columns within the current sheet.

For COBOL, we'd like to keep schema, schema loader and workbook separate, also, even though COBOL code doesn't depend on COBOL data files. We'd still like to say `schema= loader(cobol source)`.

$$S = L(c)$$

We can imagine that an application will import a workbook class and a schema loader class. It will load the schema, then open the workbook using the schema.

However.

A COBOL schema with an occurs depending on (i.e. a DDE with `variably_located == True`) will have the schema depending on each row in addition to the overall loading.

We're really taking about a Baseline Schema, S_b, and a Row-Level Schema, S_r, that is built by resolving any Occurs Depending On

$$S_b = L(c)$$

$$s_r = R(d, S_b)$$

We've changed `schema_baseline= loader(cobol source)` and then, for each row, `schema_row= setSizeAndOffset(data, schema_baseline)`.

Where To Recompute The fundamental issue is this: when can we recompute the offsets?

The choices for computing the offsets are these:

- At `COBOL_File.rows_of()` time – eagerly, but in the wrong module. See below.
- At `COBOL_File.row_get()` time – a bit more lazy, but still in the wrong module, since it's here, not in `cobol.loader`.
- In the application before doing any schema processing on a given row. Very lazy. But now the application must be more deeply involved in ODO processing. The application would do something like the following. Sadly, it has a line that's easy to overlook.

```
with open(``xyzzy.cob'') as source:
    dde_list, schema = COBOL_schema( source )
with stingray.cobol.Character_File( filename, schema=schema ) as wb:
    sheet= wb.sheet( filename )
    for row in sheet.rows():
        cobol.loader.setSizeAndOffset( dde_list[0] )
        dump( schema, row )
```

The Module Dependency Problem The Usage class properly depends on cobol. The cobol.loader.make_attr() function, also, properly depends on cobol.

The idea is that workbooks are more fundamental than schema. We might need to use one workbook to build a schema to read another workbook. Schema are higher-level constructs.

We want to avoid any circular dependency between cobol.loader referring back to cobol. The schema.RepeatingAttribute definition has a weak version of this undesirable. dependency. We finesse it by defining a bunch of properties that exploit the underlying DDE details without an explicit import of the DDE class.

To assure that cobol does not depend on cobol.loader, we'd have the class schema.RepeatingAttribute entirely built without reference to the base DDE. This, however, means that we would effectively clone the hierarchical relationships into the schema.RepeatingAttribute objects. Why bother?

If we extend COBOL_File.rows_of() or COBOL_File.row_get(), we exacerbates the problem because it would introduce a circular import. This would make cobol depend on cobol.loader explicitly.

Resolution The setSizeAndOffset() function as well as a few other post-processing functions belong in an intermediate module that both cobol and cobol.loader depend on.

Specifically, Cell definitions, DDE definitions, and the related functions required to build schema attributes from DDE's.

That way, cobol can import cobol.defs.setSizeAndOffset.

Also, cobol.loader can import cobol.defs.DDE.

And cobol.RepeatingAttribute can depend on cobol.defs.DDE.

Overheads

```
"""stingray.cobol.defs -- COBOL DDE and Tools."""
import logging
import weakref
import warnings

import stingray.cell
```

A module-level logger.

```
logger= logging.getLogger( __name__ )
```

Exception

class cobol.defs.**UnsupportedError**

We have syntax which expresses an unsupported feature of the COBOL language.

```
class UnsupportedError( Exception ):
    """A COBOL DDE has features not supported by this module."""
    pass
```

The most important unsupported feature may be "separate signs." These may be required for decoding bytes in some files.

Cell Subclasses and Conversions

Rather than tinker too much with the `cell` module, it seems better to introduce new `cell.Cell` subclasses unique to COBOL, EBCIDC and COMP-3 data.

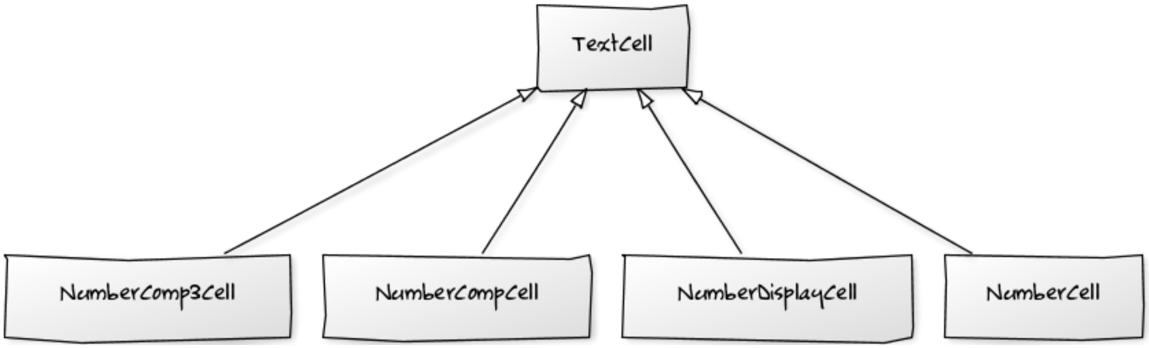
There are three relevant features.

- Proper conversion from source characters or bytes.
- Preservation of the source characters (or bytes) for creating character-level (or byte-level) structured dumps of a record.
- Preservation of the original DDE attributes, because there is so much information required to interpret the bytes.

Consequently, even the `cell.TextCell` must be extended to include preservation of raw data.

Further, we have a distinction between text and numbers which are “USAGE DISPLAY”.

```
http://yuml.me/diagram/scruffy;/class/
#cobol.cell,
[TextCell]^[NumberCell],
[NumberCell]^[NumberDisplayCell],
[NumberCell]^[NumberCompCell],
[NumberCell]^[NumberComp3Cell],
[TextCell]^[ErrorCell],
```



Important: Non-Polymorphic.

These classes are profound extensions to the base definitions of `cell`. They are not polymorphic with the base classes. COBOL processing is not transparently identical to other workbook processing.

These cells are conventionally built by the the `cobol.COBOL_File` version of Workbook as a factory. These are rarely built any other way.

class `cobol.defs.`**`TextCell`**

```
class TextCell( stingray.cell.TextCell ):
    """A COBOL TextCell, usually Usage Display."""
    def __init__( self, raw, workbook, attr ):
        self.raw, self.workbook= raw, workbook
        self._value= workbook.text( self.raw, attr )
```

class `cobol.defs.`**`NumberCell`**

This is an abstraction to simply hold all the standard conversions

```
class NumberCell( stingray.cell.NumberCell ):
    """A COBOL number."""
    def to_int( self ): return int(self.value)
    def to_float( self ): return float(self.value)
    def to_decimal( self, digits=None ): return self.value
    def to_str( self ): return str(self.value)
```

class `cobol.defs.`**`NumberDisplayCell`**

```
class NumberDisplayCell( NumberCell ):
    """A COBOL Usage Display Numeric Cell."""
    def __init__( self, raw, workbook, attr ):
        self.raw, self.workbook= raw, workbook
        self._value= workbook.number_display( self.raw, attr )
```

class `cobol.defs.`**`NumberCompCell`**

```
class NumberCompCell( NumberCell ):
    """A COBOL Usage COMP Numeric Cell.
    Three formats.  Half-word, whole-word and double-word.
    """
    def __init__( self, raw, workbook, attr ):
        self.raw, self.workbook= raw, workbook
        self._value= workbook.number_comp( self.raw, attr )
```

class `cobol.defs.`**`NumberComp3Cell`**

```
class NumberComp3Cell( NumberCell ):
    """A COBOL Usage COMP-3 Numeric Cell.."""
    def __init__( self, raw, workbook, attr ):
        self.raw, self.workbook= raw, workbook
        self._value= workbook.number_comp3( self.raw, attr )
```

class `cobol.defs.`**`ErrorCell`**

```
class ErrorCell( stingray.cell.ErrorCell ):
    """A COBOL ErrorCell, bad data bytes with no relevant value."""
    def __init__( self, raw, workbook, attr, exception=None ):
        self.raw, self.workbook= raw, workbook
        self._value= None
        self.exception= exception
    def __repr__( self ):
        return "{0}({1!r}, {2!r})".format(
            self.__class__.__name__, self.exception, self.raw )
```

Essential Class Definitions

The essential class definitions define the DDE we're attempting to build. We can separated this structure into a few high-level subject areas:

- Usage Strategy Hierarchy defines the various kinds of USAGE options.
- Allocation Strategy Hierarchy defines the relationships among DDE's: Predecessor/Successor, Group/Elementary or Redefines.
- Occurs Strategy Hierarchy defines the Occurs options of Default (no Occurs), simple Occurs, and more complex Occurs Depending On.
- The DDE Class itself.

Usage Strategy Hierarchy The `Usage` class combines information in the picture, usage, sign and synchronized clauses.

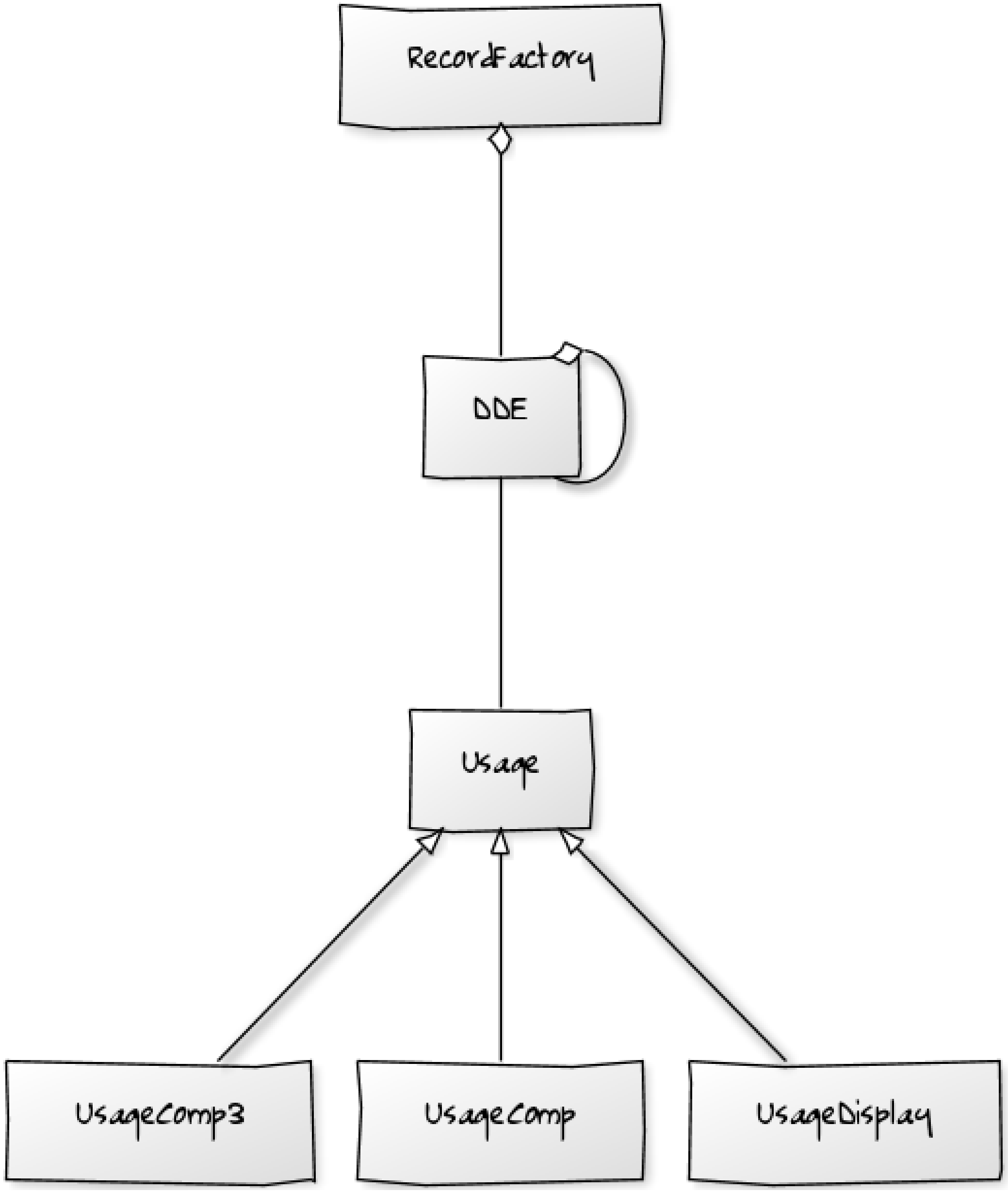
The **Strategy** design pattern allows a DDE element to delegate the `Usage.size()` and `Usage.create_func()` operations to this class.

The `Usage.size()` method returns the number of bytes used by the data element.

- For usage `DISPLAY`, the size is computed directly from the picture clause.
- For usage `COMP`, the size is 2, 4 or 8 bytes based on the picture clause.
- For usage `COMP-3`, the picture clause digits are packed two per byte with an extra half-byte for sign information. This must be rounded up. COMP-3 fields often have an odd number of digits to reflect this.

The `Usage.create_func()` method returns a `cell.Cell` type that should be built from the raw bytes.

```
http://yuml.me/diagram/scruffy;/class/
#cobol_loader_usage,
[RecordFactory]<>-[DDE],
[DDE]<>-[DDE],
[DDE]-[Usage],
[Usage]^[UsageDisplay],
[Usage]^[UsageComp]
[Usage]^[UsageComp3]
```



class `cobol.defs.`**`Usage`**

The Usage class provides detailed representation and conversion support for a given DDE. A `schema.Attribute` will refer to a `cobol.defs.DDE`. This DDE will have a `Usage` object that shows how to create the underlying `Cell` instance from the raw data in the `cobol.COBOL_File` subclass of `Workbook`.

For numeric types, this may mean a fallback from creating a `NumberCell` to creating a `ErrorCell`. If the number is invalid in some way, then an error is required.

The superclass of `Usage` is abstract and doesn't compute a proper size.

```
class Usage:
    """Covert numeric data based on Usage clause."""
    def __init__( self, source ):
        self.source_= source
        self.final= source
        self.numeric= None # is the picture all digits?
        self.length= None
        self.scale= None
        self.precision= None
        self.signed= None
        self.decimal= None
    def setTypeInfo( self, picture ):
        """Details from parsing a PICTURE clause."""
        self.final= picture.final
        self.numeric = not picture.alpha
        self.length = picture.length
        self.scale = picture.scale
        self.precision = picture.precision
        self.signed = picture.signed
        self.decimal = picture.decimal
    def source( self ):
        return self.source_
```

Usage.**create_func**()

```
def create_func( self, raw, workbook, attr ):
    """Converts bytes to a proper Cell object.
    NOTE: EBCDIC->ASCII conversion handled by the Workbook object.
    """
    return stingray.cobol.TextCell( raw, workbook, attr )
```

Usage.**size**(*picture*)

The count is in bytes. Not characters.

```
def size( self, picture ):
    """Default for group-level items."""
    return 0
```

class cobol.defs.**UsageDisplay**

Usage “DISPLAY” is the COBOL language default. It’s also assumed for group-level items.

```
class UsageDisplay( Usage ):
    """Ordinary character data which is numeric."""
    def __init__( self, source ):
        super().__init__( source )
    def create_func( self, raw, workbook, attr ):
        if self.numeric:
            try:
                return NumberDisplayCell( raw, workbook, attr )
            except Exception as e:
                error= ErrorCell( raw, workbook, attr, exception=e )
                return error
        return stingray.cobol.TextCell( raw, workbook, attr )
    def size( self ):
        """Return the actual size of this data, based on PICTURE and SIGN."""
        return len(self.final)
```

class cobol.defs.**UsageComp**

Usage “COMPUTATIONAL” is binary-encoded data.

```
class UsageComp( Usage ):
    """Binary-encoded COMP data which is numeric."""
    def __init__( self, source ):
        super().__init__( source )
    def create_func( self, raw, workbook, attr ):
        try:
            return NumberCompCell( raw, workbook, attr )
        except Exception as e:
            error= ErrorCell( raw, workbook, attr, exception=e )
            return error
    def size( self ):
        """COMP is binary half word, whole word or double word."""
        if len(self.final) <= 4:
            return 2
        elif len(self.final) <= 9:
            return 4
        else:
            return 8
```

class `cobol.defs.`**`UsageComp3`**

Usage "COMP-3" is packed-decimal encoded data.

```
class UsageComp3( Usage ):
    """Binary-Decimal packed COMP-3 data which is numeric."""
    def __init__( self, source ):
        super().__init__( source )
    def create_func( self, raw, workbook, attr ):
        try:
            return NumberComp3Cell(raw, workbook, attr)
        except Exception as e:
            error= ErrorCell( raw, workbook, attr, exception=e )
            return error
    def size( self ):
        """COMP-3 is packed decimal."""
        return (len(self.final)+2)//2
```

Allocation Strategy Hierarchy We actually have three kinds of allocation relationships among DDE items.

- Predecessor/Successor
- Group/Elementary
- Redefines

[*Formerly, we had only two subclasses.*]

This leads to a **Strategy** class hierarchy to handle the various algorithmic choices.

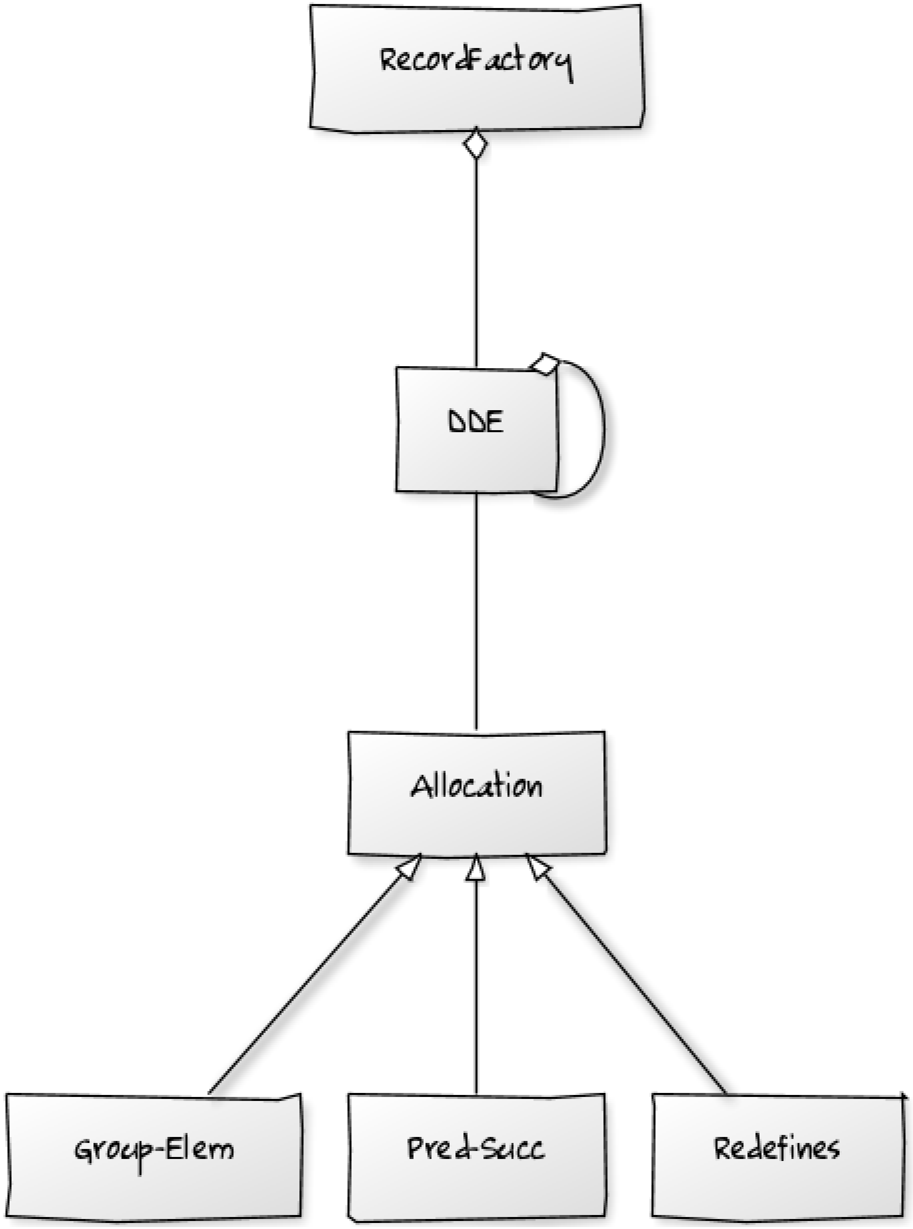
The Pred/Succ strategy computes the offset to a specific item based on the predecessor. This is the default for non-head items in a group.

The Group/Elem strategy computes the offset based on the offset to the parent group. This is the default for the head item in a group.

The Redefines strategy depends on another element: not it's immediate predecessor. This element will be assigned the same offset as the element on which it depends.

The **Strategy** design pattern allows an element to delegate the `Redefines.offset()`, and `Redefines.totalSize()` methods.

```
http://yuml.me/diagram/scruffy;/class/
#cobol_loader_redefines,
[RecordFactory]<>-[DDE],
[DDE]<>-[DDE],
[DDE]-[Allocation],
[Allocation]^[Redefines],
[Allocation]^[Pred-Succ],
[Allocation]^[Group-Elem]
```



class `cobol.defs.`**`Allocation`**

The `Allocation` superclass defines an abstract base class for the various allocation strategies.

```
class Allocation:
    def __init__( self ):
        self.dde= None
    def resolve( self, aDDE ):
        """Associate back to the owning DDE."""
        self.dde= weakref.ref(aDDE)
```

class cobol.defs.**Redefines**

The Redefines subclass depends on another element. It uses the referenced name to look up the offset and total size information.

For this to work, the name must be resolved via the Redefines.resolve() method. The resolver() function applies the Redefines.resolve() method throughout the structure.

```
class Redefines(Allocation):
    """Lookup size and offset from another field we refer to."""
    def __init__( self, name, refers_to=None ):
        super().__init__()
        self.name= name
        self.refers_to= refers_to # Used for unit testing
    def source( self ):
        return "REDEFINES {0}".format( self.refers_to.name )
```

Redefines.**resolve**(*aDDE*)

```
def resolve( self, aDDE ):
    """Search the structure for the referenced name.
    Must be done before sizing can be done.
    """
    super().resolve( aDDE )
    self.refers_to= aDDE.top().get( self.name )
```

Redefines.**offset**(*offset*)

For a redefines, this uses the resolved refers_to name and fetches the offset.

```
def offset( self, offset ):
    """:param offset: computed offset for this relative position.
    :return: named DDE element offset instead.
    """
    return self.refers_to.offset
```

Redefines.**totalSize**()

```
def totalSize( self ):
    """:return: total size of this DDE include all children and occurs.
    """
    warnings.warn("totalSize method is deprecated", DeprecationWarning )
    return 0
```

Note that 01 level items may have a REDEFINES. However, this can never meaningfully redefine anything. All 01 level definitions start at an offset of 0 by definition. A copybook may include multiple 01 levels with REDEFINES clauses; an 01-level REDEFINES is irrelevant with respect to offset and size calculations.

class cobol.defs.**Successor**

The Successor subclass does not depend on a named element, it depends on the immediate predecessor. It uses that contextual offset and size information provided by the setSizeAndOffset() function.

```
class Successor(Allocation):
    """More typical case is that the DDE follows it's predecessor.
    It's not first in a group, nor is it a redefines.
    """
    def __init__( self, pred ):
        super().__init__()
        self.refers_to= pred
    def source( self ):
        return ""
```

Successor.**offset**(*offset*)

For a successor, we use the predecessor in the `refers_to` field to track down the offset of the predecessor.

This field's offset is predecessor offset + predecessor total size.

The predecessor may have to do some thinking to get its total size or offset because of an Occurs Depending On situation.

```
def offset( self, offset ):
    """:param offset: computed offset to this point.
    :return: computed offset
    """
    return offset
```

Successor.**totalSize**()

The total size of a field with occurs depending on requires a record with live data. Otherwise, the total size is trivially computed from the DDE definition.

```
def totalSize( self ):
    """:return: total size of this DDE include all children and occurs.
    """
    warnings.warn("totalSize method is deprecated", DeprecationWarning )
    return self.dde().totalSize
```

class cobol.defs.**Group**

The `GROUP` subclass does not depend on a named element, it depends on the immediate parent group. It uses that contextual offset and size information provided by the `setSizeAndOffset()` function.

```
class Group(Allocation):
    """More typical case is that the DDE is first under a parent."""
    def __init__( self ):
        super().__init__()

    def source( self ):
        return ""
```

Group.**offset**(*offset*)

For the first item in a group, we use the group parent in the `dde` field to track down the offset of the group we're a member of.

This field's offset is the group offset, since this field is first in the group.

The group may have to do some recursive processing to get its predecessor's total size or offset because of an Occurs Depending On situation.

```
def offset( self, offset ):
    """:param offset: computed offset
    :return: computed offset
    """
    return offset
```

Group.**totalSize**()

This is essentially the same as the successor – it's merely an item within a DDE, we just track the first items separately with this subclass so that they can refer to the parent to walk up the tree.

```
def totalSize( self ):
    """:return: total size of this DDE include all children and occurs.
    """
    warnings.warn("totalSize method is deprecated", DeprecationWarning )
    return self.dde().totalSize
```

Occurs Strategy Hierarchy There are three species of Occurs clauses.

- Format 1. Fixed OCCURS with a number.
- Format 2. Variable OCCURS DEPENDING ON with a number and a name. The `resolver()` function sorts out the reference. Similar to the way REDEFINES is handled.
- Format 0. No Occurs, effectively OCCURS == 1 with no dimensionality issues.

This means that the `number` attribute must be derived EITHER from the definition or a data record. For ODO, we need to bind the definition to a record.

Important: Dependencies between DDE and Attribute

An OCCURS is a feature of the DDE. Data access, however, requires the `schema.Attribute`. There's no **direct** linkage from `cobol.defs.DDE` to `schema.Attribute`. There is linkage from `schema.Attribute` to `cobol.defs.DDE`.

It seems best to have Attribute independent of any particular source. A schema may not necessarily come from COBOL.

However.

To facilitate a DDE-oriented dump of raw data or specific fields of a COBOL file, we include a `weakref` from the DDE to the `Attribute` created from that DDE.

The overall top-most parent DDE associated with this object is `self.dde().top()`.

class `cobol.defs.`**`Occurs`**

```
class Occurs:
    """No OCCURS clause present. Data from a row is irrelevant."""
    default= True
    static= True
    def __str__( self ):
        return ""
    def resolve( self, aDDE ):
        self.dde= weakref.ref(aDDE)
    def number( self, aRow ):
        return 1
```

class `cobol.defs.`**`OccursFixed`**

```
class OccursFixed( Occurs ):
    """OCCURS n TIMES. Data from a row is irrelevant."""
    default= False
    static= True
    def __init__( self, number ):
        self._number= int(number)
    def __str__( self ):
        return "OCCURS {0}".format(self.number)
    def resolve( self, aDDE ):
        super().resolve(aDDE)
    def number( self, aRow ):
        return self._number
```

class `cobol.defs.`**`OccursDependingOn`**

```
class OccursDependingOn( Occurs ):
    """OCCURS TO n TIMES DEPENDING ON name. Data from a row is required."""
    default= False
    static= False
    def __init__( self, name, limit ):
        self.name= name
        self.limit= limit
        self.refers_to= None
        self.attr= None
    def __str__( self ):
        return "OCCURS TO {0} DEPENDING ON {1}".format(self.limit, self.name)
    def resolve( self, aDDE ):
        super().resolve(aDDE)
        self.refers_to= aDDE.top().get( self.name )
    def number( self, aRow ):
        """aRow.cell( schema.get(self.name) ) should have a numeric value."""
        if self.attr is None:
            schema_dict= dict( (a.name, a) for a in aRow.sheet.schema )
            self.attr= schema_dict[self.name]
        ## logger.debug( "Getting {0} from {1}".format(self.attr,aRow) )
        value= aRow.cell( self.attr ).to_int()
        return value
```

class `cobol.defs.`**`OccursDependingOnLimit`**

This is an extension to OccursDependingOn. It limits the ODO clause to the defined upper bound.

If we have `05 SOMETHING OCCURS 1 TO 5 TIMES DEPENDING ON X` and the value of `X` is greater than 5, the maximum defined value, 5, is used.

This entirely hypothetical as a possible fix to a problem. It's probably a Very Bad Idea, and should be removed.

See http://pic.dhe.ibm.com/infocenter/ratdevz/v8r0/index.jsp?topic=%2Fcom.ibm.ent.cbl.zos.doc%2Ftopics%2FMG%2Figymch1027.h

> "When the maximum length is used, it is not necessary to initialize the ODO object before the table receives data."
>
> "When TABLE-GROUP-1 is a receiving item, Enterprise COBOL moves the maximum number of character positions for it (450 bytes for TABLE-1 plus two bytes for ODO-KEY-1). Therefore, you need not initialize the length of TABLE-1 before moving the SEND-ITEM-1 data into the table."

Based on this (and bad data seen in the wild) we deduce that this upper limit clamping **may** be a language feature.

```
class OccursDependingOnLimit( OccursDependingOn ):
    """OCCURS TO n TIMES DEPENDING ON name. Data is required.
    This will clamp the result at the given upper limit.
    """
    def number( self, aRow ):
        value= super().number( aRow )
        if value > self.limit:
            return self.limit
        return value
```

DDE Class

class `cobol.defs.`**`DDE`**

The `DDE` class itself defines a single element (group or elementary) of a record. There are several broad areas of functionality for a DDE: (1) construction, (2) reporting and decoration, (3) processing record data.

The class definition includes the attributes determined at parse time, attributes added during decoration time and attributes used during decoration processing.

As noted above, Group-level vs. Elementary-level *could* be separate subclasses of DDE. They aren't right now, since group-level items can be used in an application program like elementary items.

A group-level item contains subsidiary DDE's and has no PICTURE clause. An elementary-level DDE is defined by having a PICTURE clause.

All group-level DDE's are effectively string-type data. An elementary-level DDE with a numeric PICTURE is numeric-type data. It can be usage display or usage computational. An elementary-level DDE with a string PICTURE is string-type data.

Occurs and Redefines can occur at any level.

Each entry is defined by the following attributes:

level COBOL level number 01 to 49, 66 or 88.

myName COBOL variable name

occurs An instance of `Occurs`. the number of occurrences. The default is 1, which we call "format 0". There are two defined formats: format 1 has a fixed number of occurrences; format 2 is the Occurs Depending On with a variable number of occurrences.

picture the exploded picture clause, with ()'s expanded

initValue any initial value provided

allocation an instance of `Allocation` used to compute the offset and total size.

usage an instance of `Usage` to delegate data conversion properly. The actual conversion is handled by the workbook.

contains the list of contained fields within a group

parent A weakref to the immediate parent DDE

top A weakref to the overall record definition DDE.

The following decorations are applied by functions that traverse the DDE structure.

sizeScalePrecision `Picture` namedtuple with details derived from parsing the PICTURE clause

size the size of an individual occurrence

The following features may have to be computed lazily if there's an Occurs Depending On. Otherwise they can be computed eagerly.

variably_located Variably Located if any this element or any child has Occurs Depending On. Otherwise (no ODO) the DDE is Statically Located. Actually, only element **after** the ODO element are variably located. But it's simpler to treat the whole record as variable.

offset offset to this field from start of record.

totalSize overall size of this item, including all occurrences.

Additionally, this item – in a way – breaks the dependencies between a `schema.Attribute` and a DDE. It's appropriate for an Attribute to depend on a DDE, but the reverse isn't proper. However, we DDE referring to an attribute anyway.

attribute weakref to the `schema.Attribute` built from this DDE.

```
class DDE:
    """A Data Description Entry.
    """
    def __init__( self, level, name, usage=None, occurs=None, redefines=None,
        initValue=None, pic=None, sizeScalePrecision=None ):
        """Build this with the results of parsing the various clauses.
        """
        self.level= level
        self.name= name
        self.occurs= occurs if occurs is not None else Occurs()
        self.picture= pic # source, prior to parsing, below.
        self.allocation= redefines # Redefines or Successor or Group
        self.usage= usage
        self.initValue= initValue

        # Parsed picture information
        self.sizeScalePrecision= sizeScalePrecision

        # Relationships
        self.indent= 0
        self.children= []
        self.top= None # must be a weakref.ref()
        self.parent= None # must be a weakref.ref()

        # Derived property from the picture clause
        self.size= self.usage.size()

        # Because of ODO, these cannot always be computed statically.
        self.offset= 0 # self.allocation.refers_to.offset + self.allocation.refers_to.total_size
        self.totalSize= 0 # self.size * self.occurs.number(aRow)

        # Derived attribute created from this DDE.
        self.attribute= None

    def __repr__( self ):
        return "{:s} {:s} {:s}".format( self.level, self.name, map(str,self.children) )
    def __str__( self ):
        oc= str(self.occurs)
        pc= " PIC {0}".format(self.picture) if self.picture else ""
        uc= " USAGE {0}".format( self.usage.source() ) if self.usage.source() else ""
        rc= self.allocation.source()
        return "{:<2s} {:<20s}{:s}{:s}{:s}{:s}.".format( self.level, self.name, rc, oc, pc, uc )
```

Construction occurs in three general steps:

1. the DDE is created,
2. source attributes are set,
3. the DDE is decorated with size, offset and other details.
4. the DDE is transformed into a `schema.Attribute`.

```
def addChild( self, aDDE ):
    """Add a substructure to this DDE.

    This is used by RecordFactory to assemble the DDE.
    """
    if aDDE.allocation:
        # has a redefines, leave it alone
        pass
    else:
        if self.children:
            aDDE.allocation= Successor( self.children[-1] )
        else:
            aDDE.allocation= Group()
    aDDE.top= self.top # Already a weakref
    aDDE.parent= weakref.ref(self)
    aDDE.indent= self.indent+1
    self.children.append( aDDE )
```

This iterator does a pre-order depth-first traversal of the subtree. This provides a single, flat list of all elements.

class DDE.**__iter__**()

```
def __iter__( self ):
    yield self
    for c in self.children:
        for dde in c:
            yield dde
```

The process of scanning a record involves methods to locate a specific field, set the occurrence index of a field, and pick bytes of a record input buffer.

We work with `"."`-separated path names through the hierarchy. This doesn't work well in the presence of OCCURS clauses and indexes. For that, we need more complex navigation.

```
def pathTo( self ):
    """Return the complete "."-delimited path to this DDE."""
    if self.parent: return self.parent().pathTo() + "." + self.name
    return self.name

def getPath( self, path ):
    """Given a "."-punctuated Path, locate the field.
    COBOL uses "of" for this.
    """
    context= self.top # In case we're not the top.
    for name in path.split('.'):
        context= self.get( name )
    return context

def get( self, name ):
    """Find the named field, and return the relevant substructure.
    :param: name of the DDE element
    :return: DDE Object
    :raises: AttributeError if field not found
    """
    found= search( self.top(), name )
    if found:
        return found
    raise AttributeError( "Field {:s} unknown in this record".format(name) )
```

Work with Occurs Depending On. The `variably_located()` question may only apply to top-level (01, parent=None) DDE's.

```
@property
def variably_located( self ):
    if self.occurs.static:
        return any( c.variably_located for c in self.children )
    return True # Not static

def setSizeAndOffset( self, aRow ):
    setSizeAndOffset( self, aRow )
```

DDE Preparation Processing

There are a number of classes (and functions) to support parsing. We need to transform the DDE's to a `schema.Schema` which is a flattened list of `schema.Attribute` objects for each element in the DDE.

There are a number of processing steps which are applied to the overall DDE. These functions depend on the DDE's ability to do it's own recursive pre-order traversal as an iterator.

- `source()`. Dumps canonical source.
- `report()`. Reports on the compiled results.
- `search()`. Searches through the hierarchy.
- `resolver()`. Resolves REDEFINES and DEPENDING ON.
- `setDimensionality()`. Propagates dimensionality down from group to elementary items.

Additionally, we want to prepare for size and offset calculation. Sometimes, there are no ODO's and we can compute these statically. Other times, there's an ODO and we can't compute size or offset without data.

`cobol.defs.`**`report`**(*top*)

Report the structure of this DDE in COBOL-like notation enriched with offsets and sizes.

```
def report( top ):
    """Report on copybook structure."""
    for aDDE in top:
        if aDDE.variably_located:
            pass # Nothing special (yet)
        if aDDE.sizeScalePrecision and not aDDE.sizeScalePrecision.alpha:
            final, alpha, length, scale, precision, signed, decimal = aDDE.sizeScalePrecision
            nSpec= '{:d}.{:d}'.format( length, precision )
        else:
            nSpec= ""
        print( "{:<65s} {:3s} {:3s} {:5s}".format(aDDE.indent*'  '+str(aDDE),
            aDDE.offset, aDDE.size, nSpec) )
```

`cobol.defs.`**`source`**(*top*)

Print a version of the source.

```
def source( top ):
    """Display a canonical version of the source from copybook parsing."""
    for aDDE in top:
        print( aDDE.indent*'  '+str(aDDE) )
```

`cobol.defs.`**`search`**(*top*, *aName*)

Search the structure for a given name. This returns the found value or `None`.

```
def search( top, aName ):
    """Search down through the copybook structure."""
    for aDDE in top:
        if aDDE.name == aName:
            return aDDE
```

`cobol.defs.`**`resolver`**(*top*)

Apply `Allocation.resolve()` throughout the structure. For each `REDEFINES` or `OCCURS DEPENDING ON` clause, locate the DDE to which it refers, saving a repeated searches.

```
def resolver( top ):
    """For each DDE.allocation which is based on REDEFINES, locate the referenced
    name.  For each DDE.occurs which has OCCURS DEPENDING ON, locate the
    referenced name.
    """
    for aDDE in top:
        aDDE.allocation.resolve( aDDE )
        aDDE.occurs.resolve( aDDE )
```

For allocation, we have three relationships: Successor, Group, and Redefines. The first two don't involve names. Only Redefines involves a name that we want to resolve before proceeding.

We *could* rely on memoization and do name resolution lazily as needed.

For occurs, we also have three versions: Default (effectively 1), Fixed, and Depends On. The first two don't involve names. Only Depends On involves a name that we want to resolve before proceeding.

cobol.defs.**setDimensionality**(*top*)

Set Dimensionality for each DDE. This will be the sequence of the non-default `OCCURS` clauses that apply to each item. This is the item's, plus any belonging to the parents of the item.

The order is from top down to the elementary item.

```
def setDimensionality( top ):
    def dimensions( aDDE ):
        """:returns: A tuple of parental DDE’s with non-1 occurs clauses."""
        if aDDE.occurs.default:
            this_level= tuple()
        else:
            this_level= tuple( (aDDE,) )
        if aDDE.parent:
            return dimensions( aDDE.parent() ) + this_level
        else:
            # Reached the top!
            return this_level
    for aDDE in top:
        aDDE.dimensionality = dimensions( aDDE )
```

Set Size and Offset

Sometimes, we can calculate field sizes and offsets statically. If `not top.variably_located` then the structure is entirely static. This function walks the structure, setting total size and offset.

Other times the locations are variable. If `top.variably_located` then the structure has at least one ODO.

Todo

refactor setSizeAndOffset()

Refactor setSizeAndOffset() into the `Allocation` class methods to remove isinstance() nonsense.

cobol.defs.**setSizeAndOffset**(*aDDE*, *aRow=None*, *base=0*)

This can be used with or without a row. The common case of all statically located items does not require a row. It must be used with a row for the case of OCCURS Depending On.

Todo

Fix performance.

This is called once per row: it needs to be simpler and faster. Some refactoring can eliminate the if statements.

```
def setSizeAndOffset( aDDE, aRow=None, base=0 ):
    """Given a top-level DDE, a Row of data (or None),
    assign offset, size, totalSize.
    Also, the case of an 88-level item, copy the parent USAGE to the child.

    size is the instance size. For non-group items, it comes from the
    PIC and OCCURS. For group items it's the sum of the children's sizes.

    totalSize = size * occurs.
    """
    # Pre-order: handle this item first.
    if isinstance(aDDE.allocation, Redefines):
        # REDEFINES simply copies details from the other item.
        # TODO: Push into Redefines
        # base= aDDE.offset= allocation.offset( base )
        base= aDDE.offset= aDDE.allocation.refers_to.offset
        aDDE.totalSize= aDDE.allocation.refers_to.totalSize
    else:
        # TODO: Push into Allocation as the generic rule.
        # base= aDDE.offset= allocation.offset( base )
        aDDE.offset= aDDE.allocation.offset(base)
        aDDE.totalSize= 0
    # Initialize the size -- it may get updated below for group-level items.
    aDDE.size= aDDE.usage.size()

    ## logger.debug( "{0} Enter {1} offset={2}".format(">"*aDDE.indent, aDDE, aDDE.offset) )

    # For non-picture group-level, handle all of the children.
    for child in aDDE.children:
        setSizeAndOffset( child, aRow, base )
        base = child.offset + child.totalSize
        if isinstance(child.allocation, Redefines):
            # TODO: Push into Redefines: does nothing.
            # aDDE.size+= allocation.size()
            pass
        else:
            # TODO: Push into Allocation as the generic rule.
            # aDDE.size+= allocation.size()
            if child.level == '88':
                pass
            else:
                aDDE.size+= child.totalSize

    # Collect final results from handling the children.
    aDDE.totalSize = aDDE.size * aDDE.occurs.number(aRow)

    ## logger.debug( "{0} Exit  {1} size={2}*{3}={4}".format(
    ##     "<"*aDDE.indent, aDDE, aDDE.size, aDDE.occurs.number(aRow), aDDE.totalSize) )
```

This function can be used during DDE load time if there are no Occurs Depending On. Otherwise, it must be used for each individual row which is read. See `sheet.LazyRow`.

1.11 The Stingray Developer's Guide

We use **Stingray** to work with data files where the schema is either external or complex (or both). We can tackle this question:

> **How do we process a file simply and consistently?**

Or, more concretely,

> **How do we make a program independent of Physical Format and Logical Layout?**

We can also use **Stingray** to answer questions about files, the schema those files purport to use, and the data on those files. Specifically, we can tackle this question:

> **How do we assure that a file and an application use the same schema?**

There are two sides to schema use:

- **Application Use.** Given a data file, we need to bind schema information to that file. For a workbook, the schema may be in the column headings of the sheet. Or it may be in a separate sheet, or a separate workbook. For a COBOL file, the schema is always in a separate COBOL-language data definition.
- **Schema Conformance.** Given a data file, how do we confirm that some schema matches the file? This is the data quality assurance question.

We do need to note the following.

> **If it was simple, we wouldn't need this package, would we?**

1.11.1 Concepts

As noted in *Introduction*, there are three levels of schema that need to be bound to a file.

- **Physical Format**. We can make this transparent to our applications. See *Workbook Package – Uniform Wrappers for Workbooks* and *The COBOL Package* for details. Everything is a workbook with a fairly limited API.
- **Logical Layout**. This is how an application program will make use of the data found in a file. Sometimes the Logical layout information can be embedded in a file: it might be the top row of a sheet in a workbook, for example. Sometimes the logical layout can be separate: a COBOL data definition, or perhaps a metadata sheet in a workbook.

  We can't make the logical layout transparent. Our applications and files both need to agree on a logical layout. The column names in the metadata, for example, must be agreed to. The order or position of the columns, however, need not be fixed.
- **Conceptual Content**. A single conceptual schema may be implemented by a number of physical formats and logical layouts. An application should be able to tolerate variability in the logical layout as long is it matches the expected **conceptual content**.

  Since we're working in Python, the conceptual schema is often a class definition. It might be a namedtuple or a SimpleNamespace, also.

We'll tackle the schema binding in several pieces.

- **File Schema**. Binding a Schema to a File describes some preliminary operational steps that make Stingray work more simply and reliably.
- **Processing**. These are the basic concepts. Data Attribute Mapping – Using a Schema, and Data Transformation.
- **Application Design Patterns**. These are more complex issues. We can then dig into **Stingray** application programming in stingray Application Design, Variant Records and COBOL REDEFINES, and Big Data Performance.

- **Data Management**. File Naming and External Schema, Binding a Schema to an Application, and Schema Version Numbering.
- **FAQ**. Some other design questions. Frequently Asked Questions.

We'll look at some demonstration programs in *Stingray Demo Applications*.

1.11.2 Binding a Schema to a File

We're only going to bind two levels of schema to a file. The conceptual schema would require some kind of formal ontology, something that's rarely available.

Logical Layout. We rely on a schema, `schema.Schema` to manage the logical layout of a file.

A workbook has two ways to bind logical layout to data values. Our `sheet.Sheet` subclass hierarchy requires a schema object. We'll load the schema using a `schema.loader` components.

- **Embedded**. This is commonly seen as column titles within the sheet. Or any variation on that theme. The common case of column titles is handled by a built-in schema loader, `schema.loader.HeadingRowSchemaLoader`. Other variations are managed by building different schema loaders.
- **External**. This is a separate sheet or separate file. In this case, we can start with `schema.loader.BareExternalSchemaLoader` to read an external schema. In the case of COBOL files, there's a separate `cobol.loader.COBOLSchemaLoader` that parses COBOL source to create a usable schema.

Physical Format. Generally, a file name provides a hint as to the physical file format. `.csv`, `.xls`, `.xlsx`, `.xlsm`, `.ods`, `.numbers` describe the physical format.

Our `cell.Cell`, `sheet.Sheet`, and `workbook.Workbook` handles many physical format details nicely.

1.11.3 Data Attribute Mapping – Using a Schema

Using a schema is the heart of the semantic problem.

We want to have just one application that is adaptable to a number of closely-related data file schemata. Ideally, there's one, but as a practical matter, there are often several similar schema.

We need to provide three pieces of information, minimally.

- Target attribute within our application.
- Target data type conversion.
- Source attribute based on attribute name or position.

We could use a number of different encodings for this relationship. We could write it as properties, or XML, or some other notation.

However, that triple is essentially a Python assignment statement with *target*, *to_type* and *source*. The simplest description is the following:

```
target = row.cell( schema['source‘] ).to_type()
```

There is a tiny bit of boilerplate in this assignment statement. When using repeating groups, items with duplicated column names, or REDEFINES clauses, the "boilerplate" expands to some additional code required to locate the source data.

For multiple attributes, this is our use case: a **Builder Function**:

```
def build_record( aRow ):
    return dict(
        name = row.cell( schema['some column'] ).to_str(),
        address = row.cell( schema['another column'] ).to_str(),
        zip = row.cell( schema['zip'] ).to_digit_str(5),
        phone = row.cell( schema['phone'] ).to_digit_str(),
    )
```

The idea is to build a single function that defines the application-specific mapping from a row in a file, given the logical layout information buried in the schema definition.

Of course, the schema can lie, and the application can misuse the data. Those are inevitable (and therefore insoluble) problems. This is why we must write customized software to handle these data sources.

In the case of variant schemata, we would like something like this.

```
def build_record_1( aRow ):
    return dict(
        name = row.cell( schema['some column'] ).to_str(),
        address = row.cell( schema['another column'] ).to_str(),
        zip = row.cell( schema['zip'] ).to_digit_str(5),
        phone = row.cell( schema['phone'] ).to_digit_str(),
    )

def build_record_2( aRow ):
    return dict(
        name = row.cell( schema['variant column'] ).to_str(),
        address = row.cell( schema['something different'] ).to_str(),
        zip = row.cell( schema['zip'] ).to_digit_str(5),
        phone = row.cell( schema['phone'] ).to_digit_str(),
    )
```

We can then define a handy factory which picks a builder based on the schema version.

make_builder(*args*)

Parameters **args** – schema version

Returns appropriate builder function for the schema

```
def make_builder( args ):
    return eval( 'build_record_{0}'.format(args.layout) )
```

The `make_builder()` function selects one of the available builders based on a run-time option.

Adding Fluency

In the case where there are no repeating groups, nor REDEFINES clauses, we can make our API slightly more fluent by building a row dictionary from row and schema. This kind of eager cell processing is risky for COBOL files. It often works, however, for well-known spreadsheet files.

```
row_dict = dict(
    (a.name, row.cell(a)) for a in schema )
```

This allows the following *target*, *to_type* and *source* triple.

```
target = row['source`].to_type()
```

This parallels the `csv` module's `DictReader` class.

1.11.4 Data Transformation

In the *Cell Module – Data Element Containers and Conversions* chapter, we noted that there are two parts to data handling: **Capture** and **Conversion**. Capture is part of parsing the physical format. Conversion is part of the final application, and has nothing to do with the schema that describes the data source.

A target data type transformation (or conversion) could be considerably more complex than the trivial case of decoding a floating-point number to a digit string. It could involve any combination of filtering, cleansing, conforming to an existing database, or rewriting.

Here’s a much more complex Builder Function.

```
def build_record_3( aRow ):
    if aRow['flag'].is_empty():
        return None
    zip_str = aRow['zip'].to_str()
    if '-' not in zip:
        if len(zip) <= 5:
            zip= aRow['zip'].to_digit_str(5)
        else:
            zip= aRow['zip'].to_digit_str(9)
    else:
        zip= zip_str.replace('-'.'')
    return dict(
        name = aRow['variant column'].to_str(),
        address = arow['different column'].to_str(),
        zip = zip,
        phone = aRow['phone'].to_digit_str(),
    )
```

This shows filtering and cleasing operations. Yes, it’s complex. Indeed, it’s complex enough that alternative domain-specific languages (i.e., properties, XLST, etc.) are much less clear than the Python.

1.11.5 Stingray Application Design

Generally, there are two kinds of testing. Conventional unit testing applies to our application to be sure the schemata are used properly. Beyond that, we have data quality testing.

For application unit testing, our programs should be decomposed into three tiers of processing.

- Row-Level. Inidividual Python objects built from one row of the input. This involves our builder functions.
- Sheet-Level. Collections of Python objects built from all rows of a sheet. This involves sheet processing functions.
- Workbook-Level. Collections of sheets.

Each of these tiers should be tested independently.

In *Unit Level Validation for Application and Data*, we’ll look at how we validate that the the input files have the expected schema. This is a kind of **Data Quality** testing. It can use the unit testing framework, but it applies to data, not applications.

Row-Level Processing

Row-level processing is centered on a suite of builder functions. These handle the detailed mapping from variant logical layouts to a single, standardized conceptual schema.

A builder function should create a simple dictionary or `types.SimpleNamespace`. Each dictionary key is the standardized attribute names used by internal Python class definitions.

Q. Why not build the final Python objects from the source row?

A. The source row needs to be validated to see if a valid object can be built. It seems simpler to map the logical layout in the source document to a standardized structure that matches the conceptual schema. This standardized structure can be validated. Then the Python object built from that structure.

This follows the design patterns from the Django project where a `ModelForm` is used to validate data before attempting to build a `Model` instance.

Here's the three-part operation: **Build, Validate and Construct**.

```
def builder_1( row ):
    return dict(
        key = row.cell( row.sheet.schema['field'] ).to_type(),
    )

def is_valid( row_dict ):
    All present or accounted for?
    return state

def construct_object( row_dict ):
    return App_Object( **row_dict )
```

The validation rules rarely change. The object construction doesn't really need to be a separate function, it can often be a simple class name.

Our sheet processing can include a function like this. We'll look at this below.

```
builder= make_builder( args )
for row in sheet:
    intermediate= builder( row )
    if is_valid(intermediate):
        yield construct_object(intermediate)
    else:
        log.error( row )
```

The builder, however, varies with the file's actual schema. We need to pick the builder based on a "logical layout" command-line option. Something like the following is used to make an application flexible with respect to layout.

```
def make_builder( args ):
    if args.layout in ("1", "D", "d"):
        return builder_1
    elif args.layout == "2":
        return builder_2
    else
        raise Exception( "Unknown layout value: {0}".format(args.layout) )
```

The builders are tested individually. They are subject to considerable change. New builders are created frequently.

The validation should be common to all logical layouts. It's not subject to much variation. The validation and object construction doesn't have the change velocity that builders have.

Configuration Options

We might want to package all builders in a separate module. This provides a focused location for change that leaves the application untouched when handling Yet Another Logical Layout.

```
def make_builder( args ):
    builder_name = 'builder_{0}'.format( args.layout )
    globals = {}
    execfile( 'builders.py', globals )
    return globals[builder_name]
```

Or

```
def make_builder( args ):
    import builders
    return eval('builders.builder_{0}'.format( args.layout ))
```

This allows a single application to be separated into invariant portions and the builders which need to be tweaked when the source file layouts change.

Sheet-Level Processing

The next higher layer is sheet-level processing. For the most part, sheets are generally rows of a single logcal type. In exceptional cases, a sheet may have multiple types coincedentally bound into a single sheet. We'll return to the multiple-types-per-sheet issue below.

For the single-type-per-sheet, we have a processing function like the following.

process_sheet (*sheet*, *builder*)

```
def process_sheet( sheet, builder=builder_1 ):
    counts= defaultdict( int )
    object_iter = (
        builder(row))
        for row in sheet.schema.rows_as_dict_iter(sheet) )
    for source in object_iter:
        counts['read'] += 1
        if is_valid(source):
            counts['processed'] += 1
            # The real processing
            obj= make_app_object( source )
            obj.save()
        else:
            counts['rejected'] += 1
    return counts
```

This kind of sheet is tested two ways. First, with a test fixture that provides specific rows based on requirements, edge-cases and other "white-box" considerations.

It is also tested with "live-file". The counts can be checked. This actually tests the file as much as it tests the sheet processing function.

Workbook Processing

The overall processing of a given workbook input looks like this.

process_workbook (*sheet*, *builder*)

```
def process_workbook( source, builder ):
    for name in source.sheets():
        # Sheet filter?  Or multi-way elif switch?
        sheet= source.sheet( name,
            sheet.EmbeddedSchemaSheet,
            loader_class=schema.loader.HeadingRowSchemaLoader )
        counts= process_sheet( sheet, builder )
        pprint.pprint( counts )
```

This makes two claims about the workbook.

- All sheets in the workbook have the same schema rules. In this example, it's an embedded schema in each sheet and the schema is the heading row. We could easily use an ExternalSchemaSheet and an external schema.
- A single `process_sheet()` function is appropriate for all sheets.

If a workbook doesn't meet these criteria, then a (potentially) more complex workbook processing function is needed. A sheet filter is usually necessary.

Sheet name filtering is also subject to the kind of change that builders are subject to. Each variant logical layout may also have a variation in sheet names. It helps to separate the sheet filter functions in the same way builders are separated. New functions are added with remarkable regularity

```
def sheet_filter_1( name ):
    return re.match( r'pattern`, name )
```

Or, perhaps something like this that uses a shell file-name pattern instead of a more sophisticated regular expression.

```
def sheet_filter_2( name ):
    return fnmatch.fnmatch( name, `pattern` )
```

Command-Line Interface

We have an optional argument for verbosity and a positional argument that provides all the files to profile.

```
def parse_args():
    parser= argparse.ArgumentParser()
    parser.add_argument( 'file', nargs='+' )
    parser.add_argument( '-l', '--layout' )
    parser.add_argument( '-v', '--verbose', dest='verbosity',
        default=logging.INFO, action='store_const', const=logging.DEBUG )
    return parser.parse_args()
```

The overall main program looks something like this.

```
if __name__ == "__main__":
    logging.basicConfig( stream=sys.stderr )
    args= parse_args()
    logging.getLogger().setLevel( args.verbosity )
    builder= make_builder( args )
    try:
        for file in args:
            with workbook.open_workbook( input ) as source:
                process_workbook( source, builder )
        status= 0
    except Exception as e:
        logging.exception( e )
        status= 3
    logging.shutdown()
    sys.exit( status )
```

This main program switch allows us to test the various functions (`process_workbook()`, `process_sheet()`, the builders, etc.) in isolation.

It also allows us to reuse these functions to build larger (and more complete) applications from smaller components.

In *Stingray Demo Applications* we’ll look at two demonstration applications, as well as a unit test.

1.11.6 Variant Records and COBOL REDEFINES

Ideally, a data source is in “first normal form”: all the rows are a single type of data. We can apply a **Build, Validate, Construct** sequence simply.

In too many cases, a data source has multiple types of data. In COBOL files, it’s common to have header records or trailer records which are summaries of the details sandwiched in the middle.

Similarly, a spreadsheet may be populated with summary rows that must be discarded or handled separately. We might, for example, write the summary to a different destination and use it to confirm that all rows were properly processed.

Because of the COBOL REDEFINES clause, there may be invalid data. We can’t assume that all attributes have valid data. This makes our processing slightly different because we can’t necessarily do eager evaluation of each row of data.

We’ll look at a number of techniques for handling variant records.

Trivial Filtering

When using an Embedded Schema Loader based on `schema.loader.HeadingRowSchemaLoader` we can extend this loader to reject rows.

The `schema.loader.HeadingRowSchemaLoader.rows()` method can do simple filtering. This is most appropriate for excluding blank rows or summary rows from a spreadsheet.

Multiple Passes and Filters

When we have multiple data types within a single sheet, we can process this data using Multiple Passes and Filters. Each pass uses different filters to separate the various types of data.

This relies on an eager production of an intermediate object from the raw data. This may not work well for COBOL files.

The multiple-pass option looks like this. Each pass applies a filter and then does the appropriate processing.

```
def process_sheet_filter_1( sheet ):
    counts= defaultdict( int )
    object_iter = (
        builder(row))
        for row in sheet.schema.rows_as_dict_iter(sheet) )
    for source in object_iter:
        counts[’read’] += 1
        if filter_1(source):
            counts[’filter_1’] += 1
            processing_1(source)
        else:
            counts[’rejected’] += 1
    return counts
```

Each filter is a simple boolean function like this.

```
def filter_1( source ):
    return some condition
```

The conditions may be trivial: `source['column'] == value`. The conditions may be more complex. It's good to encapsulate them as distinct, named functions.

We could make the filter function and processing function an argument to a generic `process_sheet()` function. Sometimes this is a good simplification, sometimes it leads to extra complexity of little value.

One Pass and Switch

When we have multiple data types within a single sheet, We can make single pass over the data, using an if-elif "switch" to distinguish the different types of rows. Each type of row is handled separately.

This relies on an eager production of an intermediate object from the raw data. This may not work well for COBOL files.

The one-pass option looks like this. A "switch" function is used to discriminate each kind of row that is found in the sheet.

```
def process_sheet_switch( sheet ):
    counts= defaultdict( int )
    object_iter = (
        builder(row))
        for row in sheet.schema.rows_as_dict_iter(sheet) )
    for source in object_iter:
        counts['read'] += 1
        if switch_1(source):
            processing_1(source)
            counts['switch_1'] += 1
        elif switch_2(source):
            processing_2(source)
            counts['switch_2'] += 1
        elif etc.
        else:
            counts['rejected'] += 1
    return counts
```

Each switch function is a simple boolean function like this.

```
def switch_1( source ):
    return some condition
```

The conditions may be trivial: `source['column'] == value`. The conditions may be more complex. It's good to encapsulate them as distinct, named functions.

We may be able to build a simple mapping from switch function to process function.

```
switch_process = (
    (switch_1, processing_1),
    (switch_2, processing_2),
)
```

This allows us to write a generic sheet processing function.

```
def process_sheet_switch( sheet, mapping ):
    counts= defaultdict( int )
    object_iter = (
        builder(row))
        for row in sheet.schema.rows_as_dict_iter(sheet) )
    for source in object_iter:
        counts['read'] += 1
        processed= None
        for switch, process in mapping:
            if switch(source):
                processed= switch.__name__
                process( source )
                counts[processed] += 1
        if not processed:
            counts['rejected'] += 1
    return counts
```

This can more easily be extended by adding to the `switch_process` mapping.

Lazy Switch Processing

The above two examples rely on building an iterator over intermediate objects. The `object_iter` builds objects eagerly. This may not always work for COBOL files. Here's a variation that might be helpful.

We'll decompose the builders so that the switch is applied first. Then the builder and processing can depend on the switch.

```
switch_build_process = (
    (switch_1, builder_1, processing_1),
    (switch_2, builder_2, processing_2),
)
```

This structure can be used with the following generic sheet processing.

```
def process_sheet_switch( sheet, mapping ):
    counts= defaultdict( int )
    for row in sheet.schema.rows(sheet):
        counts['read'] += 1
        processed= None
        for switch, builder, process in mapping:
            if switch(row):
                processed= switch.__name__
                source= builder( row )
                process( source )
                counts[processed] += 1
        if not processed:
            counts['rejected'] += 1
    return counts
```

This is slightly more complex. It the advantage of not attempting to process a row unless some initial sanity check has been done. Once the switch function determines the type of the row, then an appropriate builder can be invoked and the row processed.

In many cases, the processing starts with more complex data quality validation. If so, that can be added to the mapping. It would become a switch-builder-validator-process mapping that decomposes each step of the processing pipeline.

1.11.7 Big Data Performance

We've broken our processing down into separate steps which work with generic Python data structures. The idea is that we can use multiprocessing to spread the pipeline into separate processors or cores.

Each stage of the **Build, Validate, Construct** sequence can be decomposed. We can read raw data from the source file, apply a switch and put the raw "Row" objects into a processing queue.

A builder process can dequeue row objects from the processing queue, apply a builder, and put objects into a validation queue.

A validator process can dequeue built objects (dictionaries, for example) and validate them. Invalid objects can be written to a queue for logging. Valid objects can be written to another queue for processing.

The final processing queue will get a sequence of valid objects all of a single type. The final processing might involve (slow) database transactions, and there may need to be multiple worker processes fetching from this queue.

1.11.8 File Naming and External Schema

Some data management discipline is also essential be sure that the schema and file match up properly. Naming conventions and standardized directory structures are *essential* for working with external schema.

Well Known Formats

For well-known physical formats (`.csv`, `.xls`, `.xlsx`, `.xlsm`, `.ods`, `.numbers`) the filename extension describes the physical format. Additional information is required to determine the Logical Layout.

The schema may be loaded from column headers, in which case the binding is handled via an embedded schema loader. If the `schema.loader.HeadingRowSchemaLoader` is used, no more information is required. If a customized schema loader is used (because the headings are not trivially the first row of a sheet), then we must – somehow – bind application to external schema. The filename extension doesn't really help with this. The best choice is to use a configuration file of some kind.

If the schema is external, and we're working with a well-known physical format, then we have an even more complex binding issue. The schema will often require a customized schema loader. Each file must be associated with a schema file and a schema loader class name. File naming conventions won't help. This, too, should rely on a configuration file.

Fixed Formats and COBOL

For fixed-format files, the filename extension does not describe the physical layout. Further, a fixed format schema must combine logical layout and physical format into a single description.

For fixed format files, the following conventions help bind a file to its schema.

- The data file extension is the base name of a schema file. `mydata.someschema`. Do not use `.dat` or something else uninformative.
- Schema files must be be either a DDE file or a workbook in a well-known format. `someschema.cob` or `someschema.xlsx`.

Examples. We might see the following file names.

```
september_2001.exchange_1
november_2011.some_dde_name
october_2011.some_dde_name
exchange_1.xls
some_dde_name.cob
```

The `september_2001.exchange_1` file is a fixed format file which requires the `exchange_1.xls` metadata workbook.

The `november_2011.some_dde_name` and `october_2011.some_dde_name` files are fixed format files which require the `some_dde_name.cob` metadata.

External Schema Workbooks

A workbook with an external schema sheet must adhere to a few conventions to be usable. These rules form the basis for the `schema.loader.BareExternalSchemaLoader` class. To change the rules, extend that class.

- The standard sheet name `"Schema"` defines the appropriate sheet.
- There must be an internal meta-schema on line one of the sheet that provides the expected column names.
- The column names “name”, “offset”, “size”, “type” are used.
- Only “name” is required in general.
- For fixed format files, “offset”, “size” and “type” will also be required.
- Additional columns are allowed, but will be ignored.
- Type definitions are “text”, “number”, “date” and “boolean”.

1.11.9 Binding a Schema to an Application

We would like to be sure that our application’s expectations for a schema are aligned with the schema actually present. An application has several ways to bind its schema information.

- **Implicitly**. The code simply mentions specific columns (either by name or position).
- **Explicitly**. The code has a formal “requires” check to be sure that the schema provided by the input file actually matches the schema required by the application.

Explicit schema binding parallels the configuration management issue of module dependency. A file can be said to *provide* a given schema and an application *requires* a given schema.

Sadly, we don’t always have a pithy summary of a schema. We can’t trivially examine a file to be sure it conforms to a schema. In the case of well-known file formats with an embedded schema, we can do a test like this to determine if the schema is what we expect.

```
all( req in schema for req in ('some', 'list', 'of', 'columns') )
```

This covers 80% of the use cases. For all other cases, we don’t have a reliable way to confirm the file’s schema.

If we don’t implement this, we’re left with implicit schema binding in our applications. In short, we have to include data validity checks, a debugging log, and some kind of warning technique.

1.11.10 Schema Version Numbering

XSD’s can have version numbers. This is a very cool.

See http://www.xfront.com/Versioning.pdf for detailed discussion of how to represent schema version information.

Databases, however, lack version numbering in the schema. This leads to potential compatibilty issues between application programs that expect version 3 of the schema and an older database that implements version 2 of the schema.

Our file schema, similarly, don’t have a tidy, unambiguous numbering.

For external schema, we can embed the version in the file names. We might want to use something like this `econometrics_vendor_1.2`. This identifies the generic type of data, the source for that file, and the schema version number.

> Within a SQL database, we can easily use the schema name to carry version information. We could have a `name_version` kind of convention for all schema, allowing an application to confirm schema compatibility with a trivial SQL query.

For embedded schema, however, we have no *easy* to handle schema identification and version numbering. We're forced to build an algorithm to examine the actual names in the embedded schema to deduce the version.

This problem with embedded schema leads to using data profiling to reason out what the file is. This may devolve to a manual examination of the data profiling results to allow a human to determine the schema. Then, once the schema has been identified, command-line options can be used to bind the schema to file for correct processing.

1.11.11 Frequently Asked Questions

Junk Data

For inexplicable reasons, we can wind up with files that are damaged in some way.

> "there is a 65-byte "header" at the start of the file, what would be the best (least hacky) way to skip over the first 65 bytes?"

This is one of the reasons why use both a file name and an open file object as arguments for opening a workbook.

```
with open("file_with_junk.some_schema","rb") as cobol:
    cobol.seek( 66 )
    wb = stingray.cobol.EBCDIC_File( cobol.filename, file_object=cobol )
```

This should permit skipping past the junk.

1.12 Stingray Demo Applications

Perhaps the most important use case for **Stingray** is to write data extract applications. This is shown below in *Application Level Data Validation Technique*. We can also handle COBOL file conversions. This is shown below in *Reading COBOL Files*.

Additionally, we can use **Stingray** to provide assurance that our files and our applications both use the same conceptual schema. We need several kinds of assurance.

- How do we **confirm** the schema of a given file?
- More importantly, how do we confirm that an application can successfully use a file?

1.12.1 Unit Level Validation for Application and Data

We validate that a given file actually matches the required schema through a three-part valiation process.

1. Validate application's use of a schema via conventional unit testing. This is Unit Test The Builder Function, the first part of our testing.
2. Validate file conformance to a schema via "live-file testing". This is Live Data Test The Builder Function, the second part of our testing.
3. Validate the three-way application-schema-file binding by including a **File Validation** mode in every file processing application. We'll look at the part 3 (the 3-way binding of app-schema-file) in *Application Level Data Validation Technique*.

Of course, we must do good unit testing on the application overall. We'll assume that without further discussion. Failure to test is simply failure.

Live file testing moves beyond simple unit testing into a realm of establishing evidence that an application will process a given file correctly. We're gathering auditable historical information on file formats and contents.

Our focus here is validation of the application-schema binding as well as the file-schema binding.

We can use the `unittest` module to write tests that validate the schema shared by an application and file. There are three levels to this validation.

- Unit-level. This is a test of the builder functions more-or-less in isolation. There are two kinds of builder validation.
  - Subsets of rows. Developers usually start here.
  - Whole ("live") files. Production files are famous for having a few "odd" rows. Whole-file tests are essential.
- Integration-level. This is a test of a complete application using A test database (or collection of files) is prepared and the application actually exercised against actual files.

Not *everything* is mocked here: the "unit" is an integrated collection of components. Not an isolated class.

Unit level tests look like this.

Overheads

```
import unittest
from collections import defaultdict
import stingray.cell
import stingray.sheet
import stingray.workbook
import stingray.schema
import stingray.schema.loader
```

Builder Functions

See *The Stingray Developer's Guide* for background. We're going to need a "builder function." This transforms the source row object into the target object or collection.

Normally, we import the unit under test. It would look like this:

```
from demo.some_app import some_builder
```

For this demo, here's a sample builder function:

some_builder(*aRow*)

```
def some_builder( aRow ):
    return dict(
        key= aRow['Column "3" - string'].to_str(),
        value= aRow['Col 2.0 - float'].to_float()
    )
```

This depends on a schema that permits eager row building. That's common outside COBOL file processing.

Mock Workbook

We’ll use a mock `Workbook` to slightly simplify the builder-in-isolation test.

```
class MockWorkbook:
    def rows_of( self, sheet ):
        self.requested= sheet.name
        for r in self.mock_rows:
            yield stingray.sheet.Row( sheet, *r )
    def sheet( self, name ):
        return mock_sheet
    def row_get( self, row, attr ):
        return row[attr.position]
```

Unit Test The Builder Function

Step one is to unit test the `some_builder()` function with selected row-like objects from a the `MockWorkbook`.

There are two parts: a `schema.Schema` and a `sheet.ExternalSchemaSheet` that contains the mock row(s).

```
class Test_Builder_2( unittest.TestCase ):
    def setUp( self ):
        self.schema= stingray.schema.Schema(
            stingray.schema.Attribute( name=’Col 1 - int’ ),
            stingray.schema.Attribute( name=’Col 2.0 - float’ ),
            stingray.schema.Attribute( name=’Column "3" - string’ ),
            stingray.schema.Attribute( name="Column ’4’ - date" ),
            stingray.schema.Attribute( name=’Column 5 - boolean’ ),
            stingray.schema.Attribute( name=’Column 6 - empty’ ),
            stingray.schema.Attribute( name=’Column 7 - Error’ ), )
        self.wb= MockWorkbook()
        self.sheet= stingray.sheet.ExternalSchemaSheet( self.wb, "Test", self.schema )
        self.row= [
            stingray.cell.NumberCell(42.0, self.wb),
            stingray.cell.NumberCell(3.1415926, self.wb),
            stingray.cell.TextCell(’string’, self.wb),
            stingray.cell.FloatDateCell(20708.0, self.wb),
            stingray.cell.BooleanCell(1, self.wb),
            stingray.cell.EmptyCell(None, self.wb),
            stingray.cell.ErrorCell(’#DIV/0!’), ]
        self.wb.mock_sheet= self.sheet
        self.wb.mock_rows= [ self.row, ]
    def test_should_build_from_row( self ):
        row= next( self.sheet.rows() )
        dict_row= dict( (a.name, row.cell(a)) for a in self.schema )
        result= some_builder( dict_row )
        self.assertEqual( ’string’, result[’key’] )
        self.assertAlmostEqual( 3.1415926, result[’value’] )
```

Live Data Test The Builder Function

Step two is to unit test the `some_builder()` function with all rows in a given workbook. In this demo, we’re using `sample/excel97_workbook.xls`. Generally, we want to compute some aggregate (like a checksum) of various data items to be sure we’ve read and converted them properly.

Pragmatically, it’s sometimes hard to get a proper checksum, so we have to resort to sums, counts, and perhaps even frequency distribution tables.

```
class Test_Builder_2_Live( unittest.TestCase ):
    def setUp( self ):
        self.wb= stingray.workbook.open_workbook( "sample/excel97_workbook.xls" )
        self.sheet = stingray.sheet.EmbeddedSchemaSheet(
            self.wb, ’Sheet1’,
            stingray.schema.loader.HeadingRowSchemaLoader )
        self.schema= self.sheet.schema
    def test_should_build_all_rows( self ):
        summary= defaultdict( int )
        for row in self.sheet.rows():
            dict_row= dict( (a.name,row.cell(a)) for a in self.schema )
            result= some_builder( dict_row )
            summary[result[’key’]] += 1
        self.assertEqual( 1, summary[’string’] )
        self.assertEqual( 1, summary[’data’] )
```

Sheet-Level Testing

See *The Stingray Developer’s Guide* for background. We’re going to need a “sheet process function.” This transforms the source sheet into the target collection, usually an output file.

We can also use the unit test tools to test the application’s higher-level `process_some_sheet()` function.

Normally, we import the unit under test. It looks like this:

```
from demo.some_app import process_some_sheet
```

For this demo, here’s a sample sheet process function:

process_some_sheet(*sheet*)

```
def process_some_sheet( sheet ):
    counts= defaultdict( int )
    for row in sheet.schema.rows_as_dict_iter(sheet):
        row_dict= some_builder( row )
        counts[’key’,row_dict[’key’]] += 1
        counts[’read’] += 1
    return counts
```

The unit test checks the embedded schema and the overall row counts from processing the live file.

In this demo, we’re using `sample/excel97_workbook.xls`.

The test opens the workbook. It selects a sheet from the workbook using the class that extracts the schema from the row headers. The test then uses the `process_some_sheet()` function on the given sheet to extract data. In this case, the extraction is a frequency table.

```
class Test_Sheet_Builder_2_Live( unittest.TestCase ):
    def setUp( self ):
        self.wb= stingray.workbook.open_workbook( "sample/excel97_workbook.xls" )
        self.sheet = stingray.sheet.EmbeddedSchemaSheet(
            self.wb, ’Sheet1’,
            loader_class=stingray.schema.loader.HeadingRowSchemaLoader )
    def test_should_load_schema( self ):
        self.assertEqual( ’Col 1 - int’, self.sheet.schema[0].name )
        self.assertEqual( ’Col 2.0 - float’, self.sheet.schema[1].name )
        self.assertEqual( ’Column "3" - string’, self.sheet.schema[2].name )
        self.assertEqual( "Column ’4’ - date", self.sheet.schema[3].name )
        self.assertEqual( ’Column 5 - boolean’, self.sheet.schema[4].name )
        self.assertEqual( 'Column 6 - empty', self.sheet.schema[5].name )
        self.assertEqual( 'Column 7 - Error', self.sheet.schema[6].name )
    def test_should_build_sample_row( self ):
        counts= process_some_sheet( self.sheet )
        self.assertEqual( 2, counts['read'] )
        self.assertEqual( 1, counts['key','string'] )
        self.assertEqual( 1, counts['key','data'] )
```

Main Program Switch

This is a common unittest main program. Ideally, we'd create an actual suite object to allow combining tests.

```
if __name__ == "__main__":
    unittest.main()
```

Running the Demo

We can run this program like this:

```
python3 demo/test.py
```

This produces ordinary unitest output.

```
....
----------------------------------------------------------------------
Ran 4 tests in 0.016s

OK
```

1.12.2 Application Level Data Validation Technique

We validate that that a file actually matches a schema through a three-part valiation process. We looked at the first two parts in *Unit Level Validation for Application and Data*.

- Validate an application's use of a schema via conventional unit testing.
- Validate file conformance to a schema via "live-file testing".

In this section we'll show how to include a **File Validation** mode in every file processing application.

Having a validation mode means that we must disentangle all of the persistent state change operations from the input and processing in our application. The "normal" mode uses persistent changes based on the output. The validation mode doesn't make persistent changes; it can be viewed as a sort of "dry run": all the processing; none of the writing.

We'll do this with a combination of the **Command** and the **Strategy** design patterns. We'll create applications which validate their input file and have a simple plug-in strategy for doing any final persistent processing on that file.

State Change Commands

The **Command** design patterns is helpful for isolating state changes in an application.

Each change (create, update, delete) creates a **Command**. In validate mode, these are created but not applied. In "normal" processing mode, these are created and applied.

For Extract-Transform-Load (ETL) applications, the commands are the loads.

For create-retrieve-update-delete (CRUD) programs, the commands are variations on create, update and delete.

For data warehouse dimensional conformance applications, the command may be a slowly-changing dimension (SCD) algorithm that does insert or update (or both) into a dimension table.

For applications that involve a (potentially) complex multi-step workflow with (potentially) several state changes along the way, each change is a command.

In some cases, a fairly sophisticated **Command** class hierarchy is called for. In other cases, however, the individual commands can be merged into the validate **Strategy** object as methods.

Persistence Context Manager

One good way to distinguish between persistent and transient processing is to use a **Strategy** class hierarchy. This will have two variations on the persistent state changes.

- **Validate**. This subclass does nothing.
- **Process**. This subclass actually makes persistent state changes.

Combining the validate **Strategy** with the state change **Command** leads to class similar to the following.

The superclass does the persistent processing. This is the "normal" mode that makes proper changes to the filesystem or database.

```
class Persistent_Processing:
    stop_on_exception= True
    def __init__( self, context ):
        self.context= context
    def save_this( self, this_instance ):
        this_instance.save()
    def save_that( self, this_instance ):
        that_instance.save()
```

We'll fold in the Context Manager interface. This is a polite way to support any preparation or finalization. For example, we would use the context manager to create database connections, or finalize file system operations, etc.

```
def __enter__( self ):
    return self
def __exit__( self, exc_type, exc_val, exc_tb ):
    if exc_type is not None: return False
```

Here's a subclass which implements a safe, do-nothing strategy. This is used for "validate-mode" processing. It's a subclass that turns off persistence.

```
class Validate_Only_Processing( Persistent_Processing ):
    stop_on_exception= False
    def __init__( self, context ):
        self.context= context
    def save_this( self, this_instance ):
        pass
    def save_that( self, this_instance ):
        pass
```

Note: Alternate Design

We could revise this design to make the validation mode the superclass. The subclass could then add the persistence features.

This doesn't actually work out well in practice.

Why not?

It's too easy to overlook things in the validation mode superclass. The normal persistent processing subclass then winds up having a **lot** of extra stuff added to it.

The design winds up somewhat better looking if we remove persistence.

Having these two classes allows us to configure our application processing as follows. We can define high-level functions like `validate()` and `process()` that are identical except for the context manager that's used.

```
def validate( sheet, some_context ):
    with Validate_Only_Processing( some_context ) as mode:
        counts= process_sheet( sheet, mode )
    return counts

def process( sheet, some_context ):
    with Persistent_Processing( some_context ) as mode:
        counts= process_sheet( sheet, mode )
    return counts
```

Both of these `validate()` and `process()` functions rely on a common `process_sheeet()`. This is agnostic of the processing context; it simply does its work.

```
def process_sheet( sheet, persistence ):
    for row in sheet.schema.rows_as_dict_iter(sheet):
        try:
            this= build_this( row )
            f= ThisForm( this )
            if f.is_valid():
                persistence.save_this( this )
        except Exception, e:
            if persistence.stop_on_exception: raise
```

This allows us to effectively unit test by creating a mock version of `Persistent_Processing` and invoking the `process_sheet` function.

Example Application

We depend on a number of Python libraries. Plus, of course, we're creating workbooks, working with sheets and schema.

```
import logging
import sys
import argparse
import pprint
from collections import defaultdict

import stingray.workbook
import stingray.sheet
import stingray.schema

logger= logging.getLogger( __name__ )
```

ORM Layer

We'll often have an Object-Relational Mapping (ORM) layer. These are components that are widely shared. They could be SQLAlchemy mapped class or a Django ORM class.

It’s appropriate to supplement the ORM with a “Form” that follows the Django design pattern. This is a class that is used for validating and creating model instances.

Here’s our fake model object, `This`, and it’s form, `ThisForm`.

```
class This:
    def __init__( self, key, value ):
        self.key, self.value= key, value
    def save( self ):
        pass # The ORM save operation

class ThisForm:
    def __init__( self, **kw ):
        self.clean= kw
    def is_valid( self ):
        return self.clean[’key’] is not None
    def save( self ):
        return This( **self.clean )
```

We need to be sure that the ORM’s save operation is only invoked through our persistence processing **Strategy** object. With some libraries the persistence can be implicit, making it difficult to assure that persistence is disabled properly.

There are a number of ways to handle implicit persistence in ORM layers. It may be necessary to provide a mock database “engine” or interface in order to disable persistence.

Persistence Context Manager

These two classes define our two modes: validation and normal operations. The superclass defines the normal processing mode: we actually save objects. The subclass defines validation-only mode: we don’t save anything.

```
class Persistent_Processing:
    stop_on_exception= True
    def save_this( self, this_instance ):
        this_instance.save()
    def __enter__( self ):
        return self
    def __exit__( self, exc_type, exc_val, exc_tb ):
        if exc_type is not None: return False

class Validate_Only_Processing( Persistent_Processing ):
    stop_on_exception= False
    def save_this( self, this_instance ):
        pass
```

In larger and more sophisticated applications, there may be a much more complex set of class definitions to enable or disable persistence.

Builder Functions

See *The Stingray Developer’s Guide* for background. We’re going to need a “builder function.” This transforms the source row object into the target object or collection.

To handle variant logical layouts, a number of builder functions are provided to map the logical schemata to a more standardized conceptual schema.

```
def builder_1( row ):
    return dict(
        key= row['Column "3" - string'].to_str(),
        value= row['Col 2.0 - float'].to_float()
    )

def builder_2( row ):
    return dict(
        key= row['Column 3'].to_str(),
        value= row['Column 2'].to_float()
    )
```

Note that these builder functions are frequently added to. It's rare to get these down to a single version that always works.

Consequently, it's important to always **add** new builder functions. Logical layouts are a moving target. Old layouts don't go away; making changes to a builder is a bad idea.

It helps to have a function like this to map argument values to a builder function.

```
def make_builder( args ):
    return {
            '1': builder_1,
            '2': builder_2
            }[args.layout]
```

It can help to have a better naming convention that "_1" and "_2". In practice, however, it's sometimes hard to determine why a logical layout changed, making it hard to assign a meaningful name to the layout.

Processing

See *The Stingray Developer's Guide* for background. We're going to need a "sheet process function." This transforms the source sheet into the target collection, usually an output file.

The `process_sheet()` function is the heart of the application. This handles all the rows present in a given sheet.

```
def process_sheet( sheet, builder, persistence ):
    counts= defaultdict( int )
    for source_row in sheet.schema.rows_as_dict_iter(sheet):
        try:
            counts['read'] += 1
            row_dict= builder( source_row )
            f= ThisForm( **row_dict )
            if f.is_valid():
                counts['processed'] += 1
                this= f.save()
                persistence.save_this( this )
            else:
                counts['rejected'] += 1
        except Exception as e:
            counts['invalid'] += 1
            if persistence.stop_on_exception: raise
            summary= "{0} {1!r}".format( e.__class__.__name__, e.args )
            logger.error( summary )
            counts['error',summary] += 1
    return counts
```

Some applications will have variant processing for workbooks that contain different types of sheets. This leads to different `process_this_sheet()` and `process_that_sheet()` functions. Each will follow the above template to process all rows of the sheet.

High-Level Interfaces

These are the functions that can be used for live-file unit testing of the application as a whole. The `validate()` function uses a context manager for validation only. THe `process()` function uses the other context manager to that actual work is performed.

```
def validate( sheet, builder ):
    with Validate_Only_Processing() as mode:
        counts= process_sheet( sheet, builder, mode )
    return counts

def process( sheet, builder ):
    with Persistent_Processing() as mode:
        counts= process_sheet( sheet, builder, mode )
    return counts
```

These higher-level functions share a common `process_workbook()` function that does the real work.

```
def process_workbook( source, sheet_func, builder_func ):
    for name in source.sheets():
        logger.info( "{0} :: {1}".format( input, name ) )
        sheet= source.sheet( name,
            stingray.sheet.EmbeddedSchemaSheet,
            loader_class=stingray.schema.loader.HeadingRowSchemaLoader )
        counts= sheet_func( sheet, builder_func )
        logger.info( pprint.pformat(dict(counts)) )
```

When we do live-file testing of a given file, we can do something like the following. This uses `validate()` to assure that the file's schema is correct. See *Unit Level Validation for Application and Data* for more information.

```
from some_app import validate
class Test_Some_File( unittest.TestCase ):
    def setUp( self ):
        self.source= stingray.workbook.open_workbook( input )
        self.builder_func= builder_1
    def test_should_process_sheet1( self ):
        sheet= source.sheet( "Sheet1",
            stingray.sheet.EmbeddedSchemaSheet,
            loader_class=stingray.schema.loader.HeadingRowSchemaLoader )
        counts= validate( sheet, self.builder_func )
        self.assertEqual( 12345, counts['read'] )
```

Command-Line Interface

We have some standard arguments. While we'd like to use "-v" for validate mode, this gets confused with setting the verbosity level.

```
def parse_args():
    parser= argparse.ArgumentParser()
    parser.add_argument( 'file', nargs='+' )
    parser.add_argument( '-d', '--dry-run', default=False, action='store_true',  )
    parser.add_argument( '-l', '--layout', default='1', choices=('1','2') )
    parser.add_argument( '-v', '--verbose', dest='verbosity',
        default=logging.INFO, action='store_const', const=logging.DEBUG )
    return parser.parse_args()
```

The overall main program looks something like this. It handles a number of common main-program issues.

1. Logging.
2. Parameter Parsing. This includes interpreting options.
3. Argument Processing. This means looping over the positional arguments.
4. Opening Workbooks. Some applications can’t use the default `workbook.Opener`. A subclass of Opener, or more complex logic, may be required.
5. Gracefully catching and logging exceptions.
6. Exit Status to the OS.

```
if __name__ == "__main__":
    logging.basicConfig( stream=sys.stderr )
    args= parse_args()
    logging.getLogger().setLevel( args.verbosity )
    builder_func= make_builder( args )
    sheet_func= validate if args.dry_run else process
    logger.info( "Mode: {0}".format( sheet_func.__name__ ) )
    try:
        for input in args.file:
            with stingray.workbook.open_workbook( input ) as source:
                process_workbook( source, sheet_func, builder_func )
        status= 0
    except Exception as e:
        logging.exception( e )
        status= 3
    logging.shutdown()
    sys.exit( status )
```

Running the Demo

We can run this program like this.

```
python3 demo/app.py -d -l 1 sample/\*.csv
```

This will apply builder with layout `1` against all of the `sameple/*.csv` files.

The output looks like this

```
INFO:demo/app.py:Mode: validate
INFO:demo/app.py:sample/csv_workbook.csv :: csv_workbook
INFO:demo/app.py:{’input’: 2, ’valid’: 2}
INFO:demo/app.py:sample/simple.csv :: simple
ERROR:demo/app.py:KeyError ’Column "3" - string’
ERROR:demo/app.py:KeyError ’Column "3" - string’
ERROR:demo/app.py:KeyError ’Column "3" - string’
ERROR:demo/app.py:KeyError ’Column "3" - string’
ERROR:demo/app.py:KeyError ’Column "3" - string’
ERROR:demo/app.py:KeyError ’Column "3" - string’
ERROR:demo/app.py:KeyError ’Column "3" - string’
INFO:demo/app.py:{’KeyError \’Column "3" - string\’’: 7, ’input’: 7}
```

We can see that `sample/csv_workbook.csv` has two valid rows.

We can see that `sample/simple.csv` has seven rows, all of which are missing the required value. If all the rows are wrong, the schema in the file doesn’t match the schema required by the application.

1.12.3 Data Profiling Demonstration

This is a simple data profiling application that can be applied to workbooks to examine the data prior to creating builders.

This produces simple RST-format output on stdout.

A common use case is the following:

```
python3 demo/profile.py sample/\*.csv >profile_csv.rst
rst2html.py profile_csv.rst profile_csv.html
```

This gives us an HTML-formatted report.

Overheads

We depend on a number of Python libraries. Plus, of course, we're creating workbooks, working with sheets and schema.

```
import logging
import sys
import argparse
import pprint
from collections import defaultdict

import stingray.workbook
import stingray.sheet
import stingray.schema

logger= logging.getLogger( __name__ )
```

Processing Context

This class handles the accumulation of global statistics on the source data. This a context manager which assures that we'll successfully process the statistics in spite of any exception which might occur.

```
class Gather_Statistics:
    stop_on_exception= False
    def __init__( self ):
        self.stats= defaultdict( lambda: defaultdict(int) )
    def count( self, column, value ):
        self.stats[column][value] += 1
    def __enter__( self ):
        return self
    def __exit__( self, exc_type, exc_val, exc_tb ):
        if exc_type is not None: return False
```

See *Application Level Data Validation Technique* for a more complete example of this kind of use of a context manager.

Processing

See *The Stingray Developer's Guide* for background. We're going to need a "sheet process function." This transforms the source sheet into the target collection, usually an output file.

The `process_sheet()` function the heart of the application. This handles all the rows present in a given sheet.

We use `source_row[attr]` to accumulate the `cell.Cell` instance information. This can help identify the source data format as well as the value.

We can use `source_row[attr].value` to accumulate the "raw" Python values present in the spreadsheet.

An alternative is to use a `cell.Cell` conversion to a desired type. We might, for example, use `cell.Cell.to_str()` to convert a raw value to a string. This would better parallel the way that an application will use the data.

```
def process_sheet( sheet, persistence ):
    counts= defaultdict( int )
    for source_row in sheet.schema.rows_as_dict_iter(sheet):
        try:
            counts['read'] += 1
            for attr in source_row:
                persistence.count( attr, source_row[attr] )
        except Exception as e:
            counts['invalid'] += 1
            if persistence.stop_on_exception: raise
            summary= "{0} '{1}'".format( e.__class__.__name__, e.message )
            logger.error( summary )
            counts['error '+summary] += 1

    title= "{0} :: {1}".format( sheet.workbook.name, sheet.name )
    print( title )
    print( "="*len(title) )
    print()
    for attr in sheet.schema:
        name= attr.name
        print( name )
        print( "-"*len(name) )
        print()
        print( "..  csv-table::" )
        print()
        for k in persistence.stats[name]:
            print( '    "{0}","{1}"'.format( k, persistence.stats[name][k] ) )
        print()
    return counts
```

Some applications will have variant processing for workbooks that contain different types of sheets. This leads to different `process_this_sheet` and `process_that_sheet` functions. Each will follow the above template to process all rows of the sheet.

High-Level Interfaces

This version of `validate()` doesn't really do very much. We don't have any persistence, so there's no sensible alternative do this for simple data gathering. In more complex applications, we might have a `process()` function which does some more complex processing.

However, it's often helpful to follow the template design for other, more sophisticated, applications. For that reason, we provide the processing context as a kind of **Strategy** object to the `process_sheet()` function.

```
def validate( sheet ):
    with Gather_Statistics() as mode:
        counts= process_sheet( sheet, mode )
    return counts
```

Note that the following `process_workbook()` function makes some specific claims about the given file. In particular:

- sheet.EmbeddedSchemaSheet. The schema is embedded within each sheet.
- schema.loader.HeadingRowSchemaLoader. The schema is the heading row.

If these assumptions are not universally true, then different application programs or different process_workbook() functions must be written to handle other kinds of workbooks.

```
def process_workbook( input ):
    for name in source.sheets():
        logger.info( "{0} :: {1}".format( input, name ) )
        sheet= source.sheet( name,
            stingray.sheet.EmbeddedSchemaSheet,
            loader_class=stingray.schema.loader.HeadingRowSchemaLoader )
        counts= validate( sheet )
        logger.info( pprint.pformat(dict(counts)) )
```

Command-Line Interface

We have an optional argument for verbosity and a positional argument that provides all the files to profile.

```
def parse_args():
    parser= argparse.ArgumentParser()
    parser.add_argument( 'file', nargs='+' )
    parser.add_argument( '-v', '--verbose', dest='verbosity',
        default=logging.INFO, action='store_const', const=logging.DEBUG )
    return parser.parse_args()
```

The main-import switch allows us to import this module and reuse the components or run it as a command-line application. To run from the command line, we have several issues to address.

1. Logging.
2. Parameter Parsing. This includes interpreting options.
3. Argument Processing. This means looping over the positional arguments.
4. Opening Workbooks. Some applications can't use the default workbook.Opener. A subclass of Opener, or more complex logic, may be required.
5. Gracefully catching and logging exceptions.
6. Exit Status to the OS.

```
if __name__ == "__main__":
    logging.basicConfig( stream=sys.stderr )
    args= parse_args()
    logging.getLogger().setLevel( args.verbosity )
    try:
        for input in args.file:
            with stingray.workbook.open_workbook( input ) as source:
                process_workbook( source )
        status= 0
    except Exception as e:
        logging.exception( e )
        status= 3
    logging.shutdown()
    sys.exit( status )
```

Running the Demo

We can run this program like this:

```
python3 demo/profile.py sample/\*.csv >profile_csv.rst
rst2html.py profile_csv.rst profile_csv.html
```

The RST output file looks like this:

```
sample/csv_workbook.csv :: csv_workbook
=======================================

Col 1 - int
-----------

..  csv-table::

        "TextCell(’9973’)","1"
        "TextCell(’42’)","1"

Col 2.0 - float
---------------

..  csv-table::

        "TextCell(’3.1415926’)","1"
        "TextCell(’2.7182818’)","1"

Column "3" - string
-------------------

..  csv-table::

        "TextCell(’string’)","1"
        "TextCell(’data’)","1"

Column ’4’ - date
-----------------

..  csv-table::

        "TextCell(’09/10/56’)","1"
        "TextCell(’01/18/59’)","1"

Column 5 - boolean
------------------

..  csv-table::

        "TextCell(’TRUE’)","1"
        "TextCell(’FALSE’)","1"

Column 6 - empty
----------------

..  csv-table::

        "TextCell(’’)","2"
```

```
Column 7 - Error
----------------

..  csv-table::

        "TextCell(’#DIV/0!’)","1"
        "TextCell(’#NAME?’)","1"

sample/simple.csv :: simple
===========================

name
----

..  csv-table::

        "TextCell(’Column 6 - empty’)","1"
        "TextCell(’Column “3” - string’)","1"
        "TextCell(’Col 2.0 - float’)","1"
        "TextCell("Column ’4’ - date")","1"
        "TextCell(’Column 7 - Error’)","1"
        "TextCell(’Column 5 - boolean’)","1"
        "TextCell(’Col 1 - int’)","1"

offset
------

..  csv-table::

        "TextCell(’45’)","1"
        "TextCell(’56’)","1"
        "TextCell(’34’)","1"
        "TextCell(’67’)","1"
        "TextCell(’1’)","1"
        "TextCell(’23’)","1"
        "TextCell(’12’)","1"

size
----

..  csv-table::

        "TextCell(’11’)","7"

type
----

..  csv-table::

        "TextCell(’float’)","1"
        "TextCell(’bool’)","1"
        "TextCell(’datetime’)","1"
        "TextCell(’str’)","3"
        "TextCell(’int’)","1"
```

1.12.4 Reading COBOL Files

For sample data, we're using data found here: http://wonder.cdc.gov/wonder/sci_data/codes/fips/type_txt/cntyxref.asp

The data files are in two zip archives: http://wonder.cdc.gov/wonder/sci_data/datasets/zipctyA.zip and http://wonder.cdc.gov/wonder/sci_data/datasets/zipctyB.zip

Each of these archives contains five large files, with 2,310,000 rows of data, plus a header. The 10th file has 2,037,944 rows of data plus a header.

The member names of the ZIP archive are zipcty1 to zipcty5 and zipcty6 to zipcty10.

We'll work with two small subsets in the sample directory.

Here are the two record layouts.

```
COUNTY CROSS-REFERENCE FILE - COBOL EXAMPLE


        BLOCK CONTAINS 0 RECORDS
        LABEL RECORDS ARE STANDARD
        RECORD CONTAINS 53 CHARACTERS
        RECORDING MODE IS F
        DATA RECORDS ARE
               COUNTY-CROSS-REFERENCE-RECORD.

    01  COUNTY-CROSS-REFERENCE-RECORD.
        05   ZIP-CODE                                 PIC X(05).
        05   UPDATE-KEY-NO                            PIC X(10).
        05   ZIP-ADD-ON-RANGE.
             10  ZIP-ADD-ON-LOW-NO.
                  15  ZIP-SECTOR-NO                   PIC X(02).
                  15  ZIP-SEGMENT-NO                  PIC X(02).
             10  ZIP-ADD-ON-HIGH-NO.
                  15  ZIP-SECTOR-NO                   PIC X(02).
                  15  ZIP-SEGMENT-NO                  PIC X(02).
        05   STATE-ABBREV                             PIC X(02).
        05   COUNTY-NO                                PIC X(03).
        05   COUNTY-NAME                              PIC X(25).
```

```
COPYRIGHT HEADER RECORD - COBOL EXAMPLE


           BLOCK CONTAINS 0 RECORDS
           LABEL RECORDS ARE STANDARD
           RECORD CONTAINS 53 CHARACTERS
           RECORDING MODE IS F
           DATA RECORDS ARE
               COPYRIGHT-HEADER RECORD.

      01  COPYRIGHT-HEADER-RECORD.
          05  FILLER                                     PIC  X(05).
          05  FILE-VERSION-YEAR                          PIC  X(02).
          05  FILE-VERSION-MONTH                         PIC  X(02).
          05  COPYRIGHT-SYMBOL                           PIC  X(11).
          05  TAPE-SEQUENCE-NO                           PIC  X(03).
          05  FILLER                                     PIC  X(30).
```

Implementation

The actual COBOL code is in `demo/zipcty.cob`. This file has both record layouts. Two `01` level items with a redefines.

Here are the imports we’ll use. We’ll need the COBOL loader to read the source `demo/zipcty.cob` schema and create a COBOL file we can process.

We’ll use `types.SimpleNamespace` to build objects from the source data.

We might use `pprint` to dump values.

```
import stingray.cobol.loader
import stingray.cobol
import pprint
import types
```

When working with unknown files, we sometimes need to preview a raw dump of the records.

```
def raw_dump( schema, sheet ):
    for row in sheet.rows():
        stingray.cobol.dump( schema, row )
```

This is a handy expedient for debugging.

As suggested in *The Stingray Developer’s Guide*, here are two builder functions. The `header_builder()` function creates a header object from the first row of each `zipcty*` file.

```
def header_builder(row, schema):
    return types.SimpleNamespace(
        file_version_year= row.cell(schema['FILE-VERSION-YEAR']).to_str(),
        file_version_month= row.cell(schema['FILE-VERSION-MONTH']).to_str(),
        copyright_symbol= row.cell(schema['COPYRIGHT-SYMBOL']).to_str(),
        tape_sequence_no= row.cell(schema['TAPE-SEQUENCE-NO']).to_str(),
    )
```

The `detail_builder()` function creates a detail object from the subsequent rows of each `zipcty*` file.

Because the names within the COBOL layout are not unique at the bottom-most element level, we must use path names. The default path names include all levels of the DDE. More clever path name components might be useful here.

COBOL uses an “of” to work up the hierarchy looking for a unique name.

Maybe we could build a fluent interface `schema['ZIP-SECTOR-NO'].of('ZIP-ADD-ON-LOW-NO')`.

```
def detail_builder(row, schema):
    return types.SimpleNamespace(
        zip_code= row.cell(schema['ZIP-CODE']).to_str(),
        update_key_no= row.cell(schema['UPDATE-KEY-NO']).to_str(),
        low_sector= row.cell(schema['COUNTY-CROSS-REFERENCE-RECORD.ZIP-ADD-ON-RANGE.ZIP-ADD-ON-LOW-NO
        low_segment= row.cell(schema['COUNTY-CROSS-REFERENCE-RECORD.ZIP-ADD-ON-RANGE.ZIP-ADD-ON-LOW-N
        high_sector= row.cell(schema['COUNTY-CROSS-REFERENCE-RECORD.ZIP-ADD-ON-RANGE.ZIP-ADD-ON-HIGH-
        high_segment= row.cell(schema['COUNTY-CROSS-REFERENCE-RECORD.ZIP-ADD-ON-RANGE.ZIP-ADD-ON-HIGH
        state_abbrev= row.cell(schema['STATE-ABBREV']).to_str(),
        county_no= row.cell(schema['COUNTY-NO']).to_str(),
        county_name= row.cell(schema['COUNTY-NAME']).to_str(),
    )
```

Here’s the `process_sheet()` function which applies the builders to the various rows in each sheet. Currently, all that happens is a print of the object that was built.

Note that we’ve transformed the schema from a simple, flat list into a dictionary keyed by field name. For COBOL processing, this is essential, since the numeric order of fields isn’t often sensible.

Also note that we’ve put two versions of each name into the schema dictionary.

- The lowest level name.
- The entire path from `01` level down to the lowest level name.

[For spreadsheets, where columns are numbered, the positional information may be useful.]

```
def process_sheet( sheet ):
    schema_dict= dict( (a.name, a) for a in sheet.schema )
    schema_dict.update( dict( (a.path, a) for a in sheet.schema ) )

    counts= { ’read’: 0 }
    row_iter= sheet.rows()
    row= next(row_iter)
    header= header_builder( row, schema_dict )
    print( header )

    for row in row_iter:
        data= detail_builder( row, schema_dict )
        print( data )
        counts[’read’] += 1
    return counts
```

The top-level script must do two things:

1. Parse the `"zipcty.cob"` data definition to create a schema.
2. Open a data file as a `cobol.Character_File`. This presumes the file is all character (no COMP-3) and already translated into ASCII.

   The `process_sheet()` is applied to each file.

Here’s the script.

```
with open("sample/zipcty.cob", "r") as cobol:
    schema= stingray.cobol.loader.COBOLSchemaLoader( cobol ).load()
    #pprint.pprint( schema )
for filename in ’sample/zipcty1’, ’sample/zipcty2’:
    with stingray.cobol.Character_File( filename, schema=schema ) as wb:
        sheet= wb.sheet( filename )
        #counts= process_sheet( sheet )
        #pprint.pprint( counts )
        raw_dump( schema, sheet )
```

Running the demo

We can run this program like this:

```
python3 demo/cobol_reader.py
```

The output looks like this.

```
namespace(copyright_symbol=’ (C)USPS’, file_version_month=’09’, file_version_year=’88’, tape_sequence
namespace(county_name=’WESTCHESTER’, county_no=’119’, high_sector=’00’, high_segment=’01’, low_sector
namespace(county_name=’NORFOLK’, county_no=’021’, high_sector=’52’, high_segment=’66’, low_sector=’52
namespace(county_name=’HARTFORD’, county_no=’003’, high_sector=’49’, high_segment=’01’, low_sector=’4
namespace(county_name=’UNION’, county_no=’039’, high_sector=’22’, high_segment=’08’, low_sector=’22’,
namespace(county_name=’BRONX’, county_no=’005’, high_sector=’17’, high_segment=’05’, low_sector=’17’,
{’read’: 5}
namespace(copyright_symbol=’ (C)USPS’, file_version_month=’09’, file_version_year=’88’, tape_sequence
namespace(county_name=’SUFFOLK’, county_no=’103’, high_sector=’25’, high_segment=’43’, low_sector=’25
namespace(county_name=’CHAUTAUQUA’, county_no=’013’, high_sector=’97’, high_segment=’71’, low_sector=
namespace(county_name=’FRANKLIN’, county_no=’055’, high_sector=’90’, high_segment=’33’, low_sector=’9
namespace(county_name=’MONTGOMERY’, county_no=’091’, high_sector=’28’, high_segment=’22’, low_sector=
namespace(county_name=’WASHINGTON’, county_no=’043’, high_sector=’53’, high_segment=’05’, low_sector=
{’read’: 5}
```

Working with Archives

We don’t need to unpack the archives to work with them. We can open a ZipFile member and process that. This can be a helpful optimization when small extracts are pulled from ZIP archives.

The trick is this.

When we open the file with `stingray.cobol.Character_File( filename, schema=schema )` we can pass the file object created by `ZipFile.open()` as the second argument.

```
stingray.cobol.Character_File( filename, file_object=archive.open(filename),
schema=schema )
```

1.13 History

Latest release is 4.4.2.

1.13.1 Version 4

Version 4 dates from March, 2014. It switches to Python3.3

Change Details:

- Added a “replacing” option to the COBOL Schema Loader.

  ```
  with open("xyzzy.cob", "r") as cobol:
      schema= stingray.cobol.loader.COBOLSchemaLoader( cobol, replacing=("’WORD’", "BAR") )
  ```

  The idea is to permit a copybook that expects “REPLACING” to be parsed.

- Handle multiple 01-level declarations.

  When we look at the context for `cobol.loader.RecordFactory.makeRecord()`, this becomes an iterator which yields each `01` level DDE.

  When the stack is popped down to empty, yield the structure and then start parsing again.

  The `cobol.loader.COBOLSchemaLoader.load()` method builds the schema, iterates through all the `01` objects returned by `cobol.loader.RecordFactory.makeRecord()`, and returns `v.schema` as the resulting schema.

  Any unit test that uses `cobol.loader.RecordFactory.makeRecord()` must change to reflect it’s revised interface as an iterator.

- Add some COBOL demos.

- Handle more complex VALUES clauses for more complex 88-level items.

- Restructure cobol, cobol.loader to add a cobol.defs module.
- Handle Occurs Depending On. Parse the syntax for ODO. Update LazyRow to tweak size and offset information for each row fetched.
- Add Z/OS RECFM handling in the `cobol.EBCDIC_File` class. This will allow processing "Raw" EBCDIC files with RECFM of V and RECFM of VB – these files include BDW and RDW headers on blocks and records.
- Fix the `cobol.dump()` to properly iterate through all fields.
- Fix the `cobol.defs.Usage` hierarchy to properly handle data which can't be converted. An ErrorCell is created in the (all too common) case where the COBOL data is invalid.
- Support iWork '09 and iWork '13 Numbers Workbook files. This lead to a profound refactoring of the `stingray.workbook` module into a package.
- Remove `%` string formatting and `from __future__`. Correct class references. Replace `u''` Unicode string literals with simple string literals. https://sourceforge.net/p/stingrayreader/tickets/6/
- Updated documentation. https://sourceforge.net/p/stingrayreader/tickets/7/
- Handled precision of comp3 correctly. https://sourceforge.net/p/stingrayreader/tickets/9/
- Added `cobol.loader.Lexer_Long_Lines` to parse copybooks with junk in positions 72:80 of each line. https://sourceforge.net/p/stingrayreader/tickets/11/
- Added RECFM=N to handle variable-length files with NO BDW/RDW words. This is the default. https://sourceforge.net/p/stingrayreader/tickets/12/
- Fixed Occurs Depending On Calculation initialization error. https://sourceforge.net/p/stingrayreader/tickets/15/
- Tweaked performance slightly based on profile results.

1.13.2 Version 3

Version 3 dates from August of 2011. It unifies COBOL DDE processing with Workbook processing. They're both essentially the same thing.

The idea is to provide a schema that structures access to a file.

And release a much better version of the data profiling for COBOL files.

1.13.3 Version 2

An almost – but not quite – unrelated development was a library to unify various kinds of workbooks.

This was started in '06 or so. The context was econometric data analysis. The sources were rarely formatted consistently. While spreadsheets were common, fixed-format files (clearly produced by COBOL) had to be handled gracefully.

The misdirection of following the `csv` design patterns for eager loading of rows created small complications that were worked around badly because lazy row loading was missing from the design.

The idea of the separation of physical format from logical layout was the obvious consequence of the endless format and layout variations for a single conceptual schema.

Also, this reinforced the uselessness of trying to create a data-mapping DSL when Python expresses the data mapping perfectly.

The data mapping triple is

```
target = source.conversion()
```

Since this is – effectively – Python code, a DSL to do this is a waste of time.

1.13.4 Version 1

Version 1 started in '02 or so. Again, the context is data warehouse processing.

COBOL-based files were being examined as part of a data profiling exercise.

The data profiling use case was very simple. In effect, it was something like the following.

```
summary = defaultdict( lambda: defaultdict(int) )
def process_sheet( sheet ):
    for row in schema.rows_as_dict_iter(sheet.rows()):
        for k in row:
            summary[k][row[k]] += 1
    for attribute in summary:
        print( attribute )
        for k in summary[attribute]:
            print( k, summary[attribute][k] )
```

This version was a rewrite of the original C into Python.

It was posted into SourceForge as https://sourceforge.net/projects/cobol-dde/.

1.13.5 Version 0

Version 0 started in the late 90's. In the context of data warehouse processing, COBOL-based files were being moved from mainframe to a VAX (later a Dec Alpha).

The original processing included the following.

1. Convert the EBCDIC files from mixed display and COMP-3 to all display.
2. Copy the files from Z/OS to the VAX (or Alpha) via a magnetic tape transfer. This handled EBCDIC to ASCII conversion. (It was the 90's.)
3. Convert the resulting ASCII files from all display back to the original mixture of display and COMP-3 to resurrect the original data.
4. Process the files for warehouse loading.

The first version of this schema-based file reader did away with the painful, not-system-utility copying steps. It reduced the processing to this.

1. Copy the files from Z/OS to the VAX (or Alpha) via a magnetic tape transfer. Do no conversion. The resulting file was mixed display and COMP-3 in EBCDIC encoding.
2. Use the COBOL source DDE to determine field encoding rules. Copy the source file from mixed display and COMP-3 in EBCDIC encoding to mixed display and COMP-3 in ASCII encoding.
3. Process the files for warehouse loading.

This was written in C.

1.14 Testing

These are the various unittest modules for this package.

1.14.1 Main Test Script

This module imports the other test modules and builds a complete suite from the individual module suites.

```
"""stingray test script."""
import unittest
import sys
import logging
import test.cell
import test.sheet
import test.schema
import test.schema_loader
import test.workbook
import test.cobol
import test.cobol_loader
import test.cobol_2
import test.snappy_protobuf
```

Construction of an overall suite depends on each module providing and easy-to-use `suite()` function that returns the module's suite.

```
def suite():
    s= unittest.TestSuite()
    s.addTests( test.cell.suite() )
    s.addTests( test.sheet.suite() )
    s.addTests( test.schema.suite() )
    s.addTests( test.schema_loader.suite() )
    s.addTests( test.workbook.suite() )
    s.addTests( test.cobol.suite() )
    s.addTests( test.cobol_loader.suite() )
    s.addTests( test.cobol_2.suite() )
    s.addTests( test.snappy_protobuf.suite() )
    return s

if __name__ == "__main__":
    with test.Logger( stream=sys.stderr, level=logging.INFO ):
        unittest.TextTestRunner().run(suite())
```

1.14.2 Cell Module Tests

A Cell is the unit of data. These unit tests exercise the classes in the `cell` module. For more information, see *Cell Module – Data Element Containers and Conversions*.

Overheads

```
"""stingray.cell Unit Tests."""
import unittest
import decimal
import datetime
import stingray.cell
```

Cell Tests

Mocks for sheet and workbook to test `cell.Cell` features. A `cell.Cell` belongs to a `sheet.Sheet` and a `sheet.Sheet` belongs to a `workbook.Workbook`. We need a mock for `sheet.Sheet` and `workbook.Workbook`.

```
class CellMockWorkbook:
    def __init__( self ):
        self.datemode= 0
```

`cell.EmptyCell` is always `None`.

```
class TestEmptyCell( unittest.TestCase ):
    def setUp( self ):
        self.wb= CellMockWorkbook()
        self.cell= stingray.cell.EmptyCell('', self.wb)
    def test_should_be_empty( self ):
        self.assertTrue( self.cell.is_empty() )
    def test_should_refuse( self ):
        self.assertIsNone( self.cell.to_str() )
        self.assertIsNone( self.cell.to_int() )
        self.assertIsNone( self.cell.to_float() )
        self.assertIsNone( self.cell.to_decimal() )
        self.assertIsNone( self.cell.to_datetime() )
        self.assertIsNone( self.cell.to_digit_str() )
```

`cell.TextCell` does conversions from text to other forms. Text can represent a number of things: numbers, dates or proper string values.

In the case of a CSV file, all cells are text cells, and these conversions are very important.

In the case of XLS files or XLSX files, these conversions are less important because most cells have sensible type information.

```
class TestTextCell( unittest.TestCase ):
    def setUp( self ):
        self.wb= CellMockWorkbook()
        self.cell_numb= stingray.cell.TextCell( '123', self.wb )
        self.cell_word= stingray.cell.TextCell( 'xyz', self.wb )
        self.cell_date= stingray.cell.TextCell( '9/10/1956', self.wb )
    def test_should_be_nonempty( self ):
        self.assertFalse( self.cell_numb.is_empty() )
        self.assertFalse( self.cell_word.is_empty() )
        self.assertFalse( self.cell_date.is_empty() )
    def test_should_convert_number( self ):
        self.assertEqual( 123, self.cell_numb.to_int() )
        self.assertEqual( 123.0, self.cell_numb.to_float() )
        self.assertEqual( decimal.Decimal('123'),
            self.cell_numb.to_decimal() )
        self.assertEqual( '00123', self.cell_numb.to_digit_str(5) )
    def test_should_exception_nonnumber( self ):
        with self.assertRaises(ValueError):
            self.cell_word.to_int()
        with self.assertRaises(ValueError):
            self.cell_word.to_float()
        with self.assertRaises(decimal.InvalidOperation):
            self.cell_word.to_decimal()
        with self.assertRaises(ValueError):
            self.cell_word.to_digit_str()
    def test_should_convert_string( self ):
        self.assertEqual( ’123’, self.cell_numb.to_str() )
        self.assertEqual( ’xyz’, self.cell_word.to_str() )
        self.assertEqual( ’9/10/1956’, self.cell_date.to_str() )
    def test_should_convert_date( self ):
        self.assertEqual( datetime.datetime(1956,9,10),
            self.cell_date.to_datetime() )
    def test_should_exception_nondate( self ):
        with self.assertRaises(ValueError):
            self.cell_numb.to_datetime()
        with self.assertRaises(ValueError):
            self.cell_word.to_datetime()
```

`cell.NumberCell` does some conversions from float to other forms.

```
class TestNumberCell( unittest.TestCase ):
    def setUp( self ):
        self.wb= CellMockWorkbook()
        self.cell_numb= stingray.cell.NumberCell( 123.4, self.wb )
        self.cell_date= stingray.cell.NumberCell( 20708.0, self.wb )
    def test_should_be_nonempty( self ):
        self.assertFalse( self.cell_numb.is_empty() )
        self.assertFalse( self.cell_date.is_empty() )
    def test_should_convert_number( self ):
        self.assertEqual( 123, self.cell_numb.to_int() )
        self.assertEqual( 123.4, self.cell_numb.to_float() )
        self.assertEqual( decimal.Decimal(’123.4’),
            self.cell_numb.to_decimal(1) )
        self.assertEqual( ’00123’, self.cell_numb.to_digit_str(5) )
        self.assertEqual( ’123.4’, self.cell_numb.to_str() )
    def test_should_convert_date( self ):
        self.assertEqual( datetime.datetime(1956,9,10),
            self.cell_date.to_datetime() )
```

`cell.FloatDateCell` does some conversions from a “float that’s really a date” to other forms.

```
class TestFloatDateCell( unittest.TestCase ):
    def setUp( self ):
        self.wb= CellMockWorkbook()
        self.cell_date= stingray.cell.FloatDateCell( 20708.0, self.wb )
    def test_should_be_nonempty( self ):
        self.assertFalse( self.cell_date.is_empty() )
    def test_should_convert_number( self ):
        self.assertEqual( 20708, self.cell_date.to_int() )
        self.assertEqual( 20708.0, self.cell_date.to_float() )
        self.assertEqual( decimal.Decimal(’20708.0’),
            self.cell_date.to_decimal(1) )
        self.assertEqual( ’20708’, self.cell_date.to_digit_str(5) )
        self.assertEqual( ’20708.0’, self.cell_date.to_str() )
    def test_should_convert_date( self ):
        self.assertEqual( datetime.datetime(1956,9,10),
            self.cell_date.to_datetime() )
```

`cell.BooleanCell` does some conversions from an “int that’s really a true/false” to other forms.

```
class TestBooleanCell( unittest.TestCase ):
    def setUp( self ):
        self.wb= CellMockWorkbook()
        self.cell_true= stingray.cell.BooleanCell( 1, self.wb )
        self.cell_false= stingray.cell.BooleanCell( 0, self.wb )
    def test_should_be_nonempty( self ):
        self.assertFalse( self.cell_true.is_empty() )
        self.assertFalse( self.cell_false.is_empty() )
    def test_should_convert_number( self ):
        self.assertEqual( 1, self.cell_true.to_int() )
        self.assertEqual( 1.0, self.cell_true.to_float() )
        self.assertEqual( decimal.Decimal(’1.0’),
            self.cell_true.to_decimal(1) )
        self.assertEqual( ’00001’, self.cell_true.to_digit_str(5) )
        self.assertEqual( ’1’, self.cell_true.to_str() )
        self.assertEqual( 0, self.cell_false.to_int() )
        self.assertEqual( 0.0, self.cell_false.to_float() )
        self.assertEqual( decimal.Decimal(’0.0’),
            self.cell_false.to_decimal(1) )
        self.assertEqual( ’00000’, self.cell_false.to_digit_str(5) )
        self.assertEqual( ’0’, self.cell_false.to_str() )
    def test_should_convert_date( self ):
        with self.assertRaises(ValueError):
            self.cell_true.to_datetime()
        with self.assertRaises(ValueError):
            self.cell_false.to_datetime()
```

`cell.ErrorCell` doesn’t do many conversions. Mostly, it raises `ValueError` when converted to anything other than a string.

```
class TestErrorCell( unittest.TestCase ):
    def setUp( self ):
        self.wb= CellMockWorkbook()
        self.cell_div0= stingray.cell.ErrorCell( ’#DIV/0!’, self.wb )
    def test_should_be_nonempty( self ):
        self.assertFalse( self.cell_div0.is_empty() )
    def test_should_convert_string( self ):
        self.assertEqual( ’#DIV/0!’, self.cell_div0.to_str() )
    def test_should_convert_number( self ):
        with self.assertRaises(ValueError):
            self.cell_div0.to_int()
        with self.assertRaises(ValueError):
            self.cell_div0.to_float()
        with self.assertRaises(ValueError):
            self.cell_div0.to_decimal(1)
        with self.assertRaises(ValueError):
            self.cell_div0.to_digit_str(5)
    def test_should_convert_date( self ):
        with self.assertRaises(ValueError):
            self.cell_div0.to_datetime()
```

`cell.DateCell` does some conversions from a proper date to other forms. A proper date doesn’t arrive from XLS or XLSX, but can arrive from CSV or fixed-format sources.

```
class TestDateCell( unittest.TestCase ):
    def setUp( self ):
        self.wb= CellMockWorkbook()
        now= datetime.datetime(1956, 9, 10, 0, 0, 0)
        self.cell_date= stingray.cell.DateCell( now, self.wb )
    def test_should_be_nonempty( self ):
        self.assertFalse( self.cell_date.is_empty() )
    def test_should_convert_number( self ):
        self.assertEqual( 20708, self.cell_date.to_int() )
        self.assertEqual( 20708.0, self.cell_date.to_float() )
        self.assertEqual( decimal.Decimal('20708.0'),
            self.cell_date.to_decimal(1) )
        self.assertEqual( '20708', self.cell_date.to_digit_str(5) )
        self.assertEqual( '1956-09-10 00:00:00', self.cell_date.to_str() )
    def test_should_convert_date( self ):
        self.assertEqual( datetime.date(1956,9,10),
            self.cell_date.to_datetime().date() )
```

Other Conversions

```
class TestDateFromString( unittest.TestCase ):
    def test_date_conversion( self ):
        convert= stingray.cell.date_from_string( "%m/%d/%Y" )
        dt= convert( "9/10/1956" )
        self.assertEqual( datetime.datetime(1956, 9, 10), dt )
    def test_datecell_conversion( self ):
        self.wb= CellMockWorkbook()
        create= stingray.cell.datecell_from_string( "%Y-%m-%d" )
        dc= create( "1956-09-10", self.wb )
        self.assertEqual( datetime.datetime(1956, 9, 10), dc.to_datetime() )
        self.assertEqual( 20708, dc.to_int() )
```

Test Suite and Runner

In case we want to build up a larger test suite, we avoid doing any real work unless this is the main module being executed.

```
import test
suite= test.suite_maker( globals() )

if __name__ == "__main__":
    print( __file__ )
    unittest.TextTestRunner().run(suite())
```

1.14.3 Sheet Module Tests

A Sheet is a collection of Cells, with a structure imposed by a Schema. These unit tests exercise the classes in the `sheet` module. For more information, see *Sheet Module – Sheet and Row Access*.

Overheads

```
"""stingray.sheet Unit Tests."""
import unittest
import decimal
import datetime
import stingray.cell
import stingray.sheet
```

Sheet Tests

The top-level `sheet.Sheet` simply produces row-as-list from a workbook.

First, we define a :py:class‘MockWorkbook‘ class to implement a minimal interface that a sheet.Sheet can rely on.

```
class MockWorkbook:
    def rows_of( self, sheet ):
        self.requested= sheet.name
        for r in self.rows:
            yield stingray.sheet.Row( sheet, *r )
    def sheet( self, name ):
        return self.mock_sheet
```

Given that, we can define a sensible unit test.

```
class TestSheet( unittest.TestCase ):
    def setUp( self ):
        self.wb= MockWorkbook( )
        self.sheet= stingray.sheet.Sheet( self.wb, 'The_Name' )
        self.wb.rows = [
            [ stingray.cell.NumberCell(1, self.wb),
              stingray.cell.TextCell("First", self.wb) ],
            [ stingray.cell.NumberCell(2, self.wb),
              stingray.cell.TextCell("Second", self.wb), ],
            ]
    def test_should_iterate( self ):
        row_list= list( self.sheet.rows() )
        self.assertEqual( 'The_Name', self.wb.requested )
        self.assertEqual( 2, len(row_list) )
        self.assertTrue( all( isinstance(r,stingray.sheet.Row) for r in row_list ) )
        row= row_list[0]
        self.assertEqual( 1, row[0].to_int() )
        self.assertEqual( "First", row[1].to_str() )
        row= row_list[1]
        self.assertEqual( 2, row[0].to_int() )
        self.assertEqual( "Second", row[1].to_str() )
    def test_should_have_attributes( self ):
        self.assertEqual( 'The_Name', self.sheet.name )
```

Eager Row Tests

An eager row is just a tuple, created by a workbook when requested by a sheet.

```
class MockSheet:
    def __init__( self, workbook, name, schema ):
        self.workbook= workbook
        self.name= name
        self.schema= schema
    def rows( self ):
        return self.workbook.rows_of( self )
```

An eager Row can be built by many of the worksheets where the physical format provides guidance on field structure and data type conversions.

```
class TestEagerRow( unittest.TestCase ):
    def setUp( self ):
        self.wb= MockWorkbook( )
        self.sheet= MockSheet( self.wb, 'The_Name', None )
        self.wb.rows = [
            ( stingray.cell.NumberCell(1, self.wb),
              stingray.cell.TextCell("First", self.wb), ),
            ( stingray.cell.NumberCell(2, self.wb),
              stingray.cell.TextCell("Second", self.wb), ),
            ]
    def test_should_iterate_eager( self ):
        row_list= list( self.sheet.rows() )
        self.assertEqual( 'The_Name', self.wb.requested )
        self.assertEqual( 2, len(row_list) )
        self.assertTrue( all( isinstance(r,stingray.sheet.Row) for r in row_list ) )
        row= row_list[0]
        self.assertEqual( 1, row[0].to_int() )
        self.assertEqual( "First", row[1].to_str() )
        row= row_list[1]
        self.assertEqual( 2, row[0].to_int() )
        self.assertEqual( "Second", row[1].to_str() )
```

Lazy Row Tests

An Lazy Row must be built by Fixed format and COBOL format. The physical format provides zero guidance on field structure and data type conversions.

```
class MockLazyWorkbook:
    def rows_of( self, sheet ):
        self.requested= sheet.name
        for r in self.rows:
            yield stingray.sheet.LazyRow( sheet, data=r )
    def sheet( self, name ):
        return self.mock_sheet
    def row_get( self, row, attribute ):
        return row._state['data'][attribute.position]

class MockSchema(list):
    def __init__( self, *args, **kw ):
        super().__init__( args )
        self.info= kw

class MockAttribute:
    def __init__( self, **kw ):
        self.__dict__.update( kw )

class TestLazyRow( unittest.TestCase ):
    def setUp( self ):
        self.schema= MockSchema(
            MockAttribute( name="col1", position=0 ),
            MockAttribute( name="col2", position=1 ),
            dde=[]
        )
        self.wb= MockLazyWorkbook( )
        self.sheet= MockSheet( self.wb, 'The_Name', self.schema )
        self.wb.rows = [
            ( stingray.cell.NumberCell(1, self.wb),
              stingray.cell.TextCell("First", self.wb), ),
            ( stingray.cell.NumberCell(2, self.wb),
              stingray.cell.TextCell("Second", self.wb), ),
            ]
    def test_should_iterate_lazy( self ):
        row_list= list( self.sheet.rows() )
        self.assertEqual( 'The_Name', self.wb.requested )
        self.assertEqual( 2, len(row_list) )
        self.assertTrue( all( isinstance(r,stingray.sheet.LazyRow) for r in row_list ) )
        row= row_list[0]
        self.assertEqual( 1, row[0].to_int() )
        self.assertEqual( "First", row[1].to_str() )
        row= row_list[1]
        self.assertEqual( 2, row[0].to_int() )
        self.assertEqual( "Second", row[1].to_str() )
```

Test Suite and Runner

In case we want to build up a larger test suite, we avoid doing any real work unless this is the main module being executed.

```
import test
suite= test.suite_maker( globals() )

if __name__ == "__main__":
    print( __file__ )
    unittest.TextTestRunner().run(suite())
```

1.14.4 Schema Module Tests

A Schema is little more than a list of Attributes, and an Attribute is little more than a named tuple.

At Attribute, however, is used to build `cell.Cell` instances.

See *Schema Package – Schema and Attribute Definitions*.

Overheads

```
"""stingray.schema Unit Tests."""
import unittest
import datetime
import stingray.schema
```

Mocks

A `MockDefaultCell` allows us to make sure that `schema.Attribute` builds only this mock object.

```
class MockDefaultCell:
    def __init__( self, value, workbook ):
        self.value= value
        self.workbook= workbook
    def __eq__( self, other ):
        return all( (
            self.__class__ == other.__class__,
            self.value == other.value,
            ) )
    def __ne__( self, other ):
        return not self.__eq__( other )

class MockNonDefaultCell( MockDefaultCell ):
    pass

class SchemaMockWorkbook:
    def __init__( self ):
        self.datemode= 0
```

Attribute Features

Attributes, generally, have names. They can have other parameters, but the attributes are generally ignored except for COBOL and Fixed format files.

We subclass `schema.Attribute` to make it depend on `MockDefaultCell` instead of some working class in `cell`.

```
class AttributeFixture( stingray.schema.Attribute ):
    default_cell= MockDefaultCell

class TestAttribute( unittest.TestCase ):
    def setUp( self ):
        self.simple= AttributeFixture(
            name="some column" )
        self.complex= AttributeFixture(
            name="column", offset=0, size=5, create=MockNonDefaultCell )
    def test_should_have_names( self ):
        self.assertEqual( "some column", self.simple.name )
        self.assertEqual( "column", self.complex.name )
```

Baseline Attribute Conversion

Attributes can also have a “create” option to create a given type of `cell.Cell` from input that’s not easily decoded. based on the physical format.

Fixed and COBOL format files are the prime example of cells that require conversions. A CSV or TAB file could fall into this category, also.

This is an essential feature of the `workbook.Fixed_Workbook`.

```
class TestAttributeConversion( unittest.TestCase ):
    def setUp( self ):
        self.simple= AttributeFixture(
            name="some column", offset= 0, size= 5 )
        self.complex= AttributeFixture(
            name="column", offset=5, size=5, create=MockNonDefaultCell )
        self.data = "12345abcde"
    def test_should_extract( self ):
        col= self.simple
        data= self.data
        s= data[col.offset:col.offset+col.size]
        self.assertEqual( "12345", s )
        col= self.complex
        data= self.data
        c= data[col.offset:col.offset+col.size]
        self.assertEqual( "abcde", c )
    def test_should_convert( self ):
        wb= SchemaMockWorkbook()
        col= self.simple
        data= self.data
        s= col.create( data[col.offset:col.offset+col.size], wb )
        self.assertEqual( MockDefaultCell("12345",wb), s )
        col= self.complex
        data= self.data
        c= col.create( data[col.offset:col.offset+col.size], wb )
        self.assertEqual( MockNonDefaultCell("abcde",wb), c )
        self.assertNotEqual( MockDefaultCell("abcde",wb), c )
```

Schema

A `schema.Schema` is essentially a list of attributes. Order matters, since names can (and frequently are) duplicates.

```
class TestSchema( unittest.TestCase ):
    def setUp( self ):
        self.schema = stingray.schema.Schema(
            AttributeFixture(
                name="some column" ),
            AttributeFixture(
                name="column", offset=0, size=5, create=MockNonDefaultCell ),
        )
    def test_should_have_names( self ):
        names = [ "some column", "column" ]
        self.assertEqual( names, [ a.name for a in self.schema ] )
```

Test Suite and Runner

In case we want to build up a larger test suite, we avoid doing any real work unless this is the main module being executed.

```
import test
suite= test.suite_maker( globals() )

if __name__ == "__main__":
    print( __file__ )
    unittest.TextTestRunner().run(suite())
```

1.14.5 Schema Loader Testing

A Schema Loader will build a schema from a variety of sources.

- Embedded in the first row of a sheet. The `schema.loader.HeadingRowSchemaLoader` handles this.
- Embedded elsewhere in a sheet. A subclass of `schema.loader.SchemaLoader` can handle this.
- An external schema, read from a separate file. The `schema.loader.ExternalSchemaLoader` or a subclass can handle this.
- A "hard-coded" schema, written as a Python object definition. A static instance of `schema.Schema` handles this gracefully.

For the essential implementation, see *Schema Loader Module – Load Embedded or External Schema*.

Also, we will have schemata written in COBOL notation, but that's handled separately.

Overheads

```
"""stingray.schema.loader Unit Tests."""
import unittest
import stingray.schema
import stingray.sheet
import stingray.schema.loader
```

First, we define mock classes.

MockWorkbook class implements a minimal interface that a sheet.Sheet can rely on.

```
class SheetMockWorkbook:
    def rows_of( self, sheet ):
        self.requested= sheet.name
        return iter( self.rows )
    def sheet( self, name, sheet_type, **kw ):
        return self.mock_sheet
```

MockSheet class implements a minimal interface that a sheet.Row can rely on.

```
class MockSheet:
    def __init__( self, workbook, sheet_name ):
        self.workbook, self.name= workbook, sheet_name
    def schema( self ):
        return self.mock_schema
    def rows( self ):
        return self.workbook.rows_of( self )
```

MockRow class implements a minimal interface that collects cell instances.

```
class MockRow:
    def __init__( self, sheet, *data ):
        self.sheet= sheet
        self.data= data
    def cell( self, attribute ):
        return self.data[attribute.position]
    def __iter__( self ):
        return iter(self.data)
```

HeadingRowSchemaLoader Tests

A schema.loader.SchemaLoader` is abstract, and doesn't require much independent testing.

A schema.loader.HeadingRowSchemaLoader builds a schema.Schema from the first row from a sheet.Sheet.

```
class TestHeadingRowSchemaParser( unittest.TestCase ):
    def setUp( self ):
        self.wb= SheetMockWorkbook( )
        self.sheet= MockSheet( self.wb, 'The_Name' )
        self.wb.rows = [
            MockRow( self.sheet, stingray.cell.TextCell("Col 1", self.wb),
              stingray.cell.TextCell("Col 2", self.wb) ),
            MockRow( self.sheet, stingray.cell.NumberCell(1, self.wb),
              stingray.cell.TextCell("First", self.wb) ),
            MockRow( self.sheet, stingray.cell.NumberCell(2, self.wb),
              stingray.cell.TextCell("Second", self.wb), ),
            ]
        self.parser= stingray.schema.loader.HeadingRowSchemaLoader( self.sheet )
    def test_should_parse( self ):
        schema = self.parser.schema()
        self.assertEqual( 2, len(schema) )
        self.assertEqual( "Col 1", schema[0].name )
        self.assertEqual( "Col 2", schema[1].name )
        rows = list( self.parser.rows() )
        self.assertEqual( 2, len(rows) )
```

ExternalSchemaLoader

An `schema.loader.ExternalSchemaLoader` builds a `schema.Schema` from another `workbook.Workbook`.

```
class TestExternalSchemaLoader( unittest.TestCase ):
    def setUp( self ):
        self.wb= SheetMockWorkbook( )
        self.wb.mock_sheet= MockSheet( self.wb, 'The_Name' )
        self.wb.mock_sheet.schema = stingray.schema.Schema(
            stingray.schema.Attribute("name",create=stingray.cell.TextCell),
            stingray.schema.Attribute("offset",create=stingray.cell.NumberCell),
            stingray.schema.Attribute("size",create=stingray.cell.NumberCell),
            stingray.schema.Attribute("type",create=stingray.cell.TextCell),
        )
        sheet= self.wb.mock_sheet
        self.wb.rows = [
            #MockRow( sheet, stingray.cell.TextCell("name", self.wb),
            #  stingray.cell.TextCell("offset", self.wb),
            #  stingray.cell.TextCell("size", self.wb),
            #  stingray.cell.TextCell("type", self.wb),
            #  ),
            MockRow( sheet, stingray.cell.TextCell("Col 1", self.wb),
              stingray.cell.NumberCell(1, self.wb),
              stingray.cell.NumberCell(5, self.wb),
              stingray.cell.TextCell("str", self.wb ),
              ),
            MockRow( sheet, stingray.cell.TextCell("Col 2", self.wb),
              stingray.cell.NumberCell(6, self.wb),
              stingray.cell.NumberCell(10, self.wb),
              stingray.cell.TextCell("str", self.wb),
              ),
            ]

        self.loader= stingray.schema.loader.ExternalSchemaLoader( self.wb, 'The_Name',)
    def test_should_parse( self ):
        schema = self.loader.schema()
        self.assertEqual( 2, len(schema) )
        self.assertEqual( "Col 1", schema[0].name )
        self.assertEqual( "Col 2", schema[1].name )
```

ExternalSchemaSheet Tests

The `sheet.ExternalSchemaSheet` creates a schema from an external source, usually another Workbook.

An external schema can be loaded using an `schema.loader.ExternalSchemaLoader`. However, we can also hard-code a Schema. Or build it some other way.

A workbook.Workbook uses this type to build each individual sheet.Sheet, attaching the schema (and column titles, etc.) to each sheet.Row that gets built.

```
class TestExternalSchemaSheet( unittest.TestCase ):
    def setUp( self ):
        self.wb= SheetMockWorkbook( )
        schema = stingray.schema.Schema(
            stingray.schema.Attribute( "Col 1" ),
            stingray.schema.Attribute( "Col 2" ),
        )
        self.sheet= stingray.sheet.ExternalSchemaSheet( self.wb, 'The_Name', schema=schema )
        self.wb.rows = [
            MockRow( self.sheet, stingray.cell.NumberCell(1, self.wb),
              stingray.cell.TextCell("First", self.wb) ),
            MockRow( self.sheet, stingray.cell.NumberCell(2, self.wb),
              stingray.cell.TextCell("Second", self.wb), ),
            ]
    def test_should_iterate( self ):
        rows_as_dict_iter = (
            dict( zip( (a.name for a in self.sheet.schema), r ) )
            for r in self.sheet.rows() )
        row_list= list(  rows_as_dict_iter )
        self.assertEqual( 'The_Name', self.wb.requested )
        self.assertEqual( 2, len(row_list) )
        self.assertTrue( all( isinstance(r,dict) for r in row_list ) )
        row= row_list[0]
        self.assertEqual( 1, row['Col 1'].to_int() )
        self.assertEqual( "First", row['Col 2'].to_str() )
        row= row_list[1]
        self.assertEqual( 2, row['Col 1'].to_int() )
        self.assertEqual( "Second", row['Col 2'].to_str() )
    def test_should_have_attributes( self ):
        self.assertEqual( 'The_Name', self.sheet.name )
```

EmbeddedSchemaSheet Tests

An sheet.EmbeddedSchemaSheet creates a schema from the workbook and produces rows. It relies on a schema.loader.HeadingRowSchemaLoader, which we must mock.

Here's a mock schema loader.

```
class MockLoader:
    def __init__( self, sheet ):
        self.sheet= sheet
    def schema( self ):
        schema= stingray.schema.Schema(
            stingray.schema.Attribute( "Col 1" ),
            stingray.schema.Attribute( "Col 2" )
            )
        return schema
    def rows( self ):
        row_iter= iter( self.sheet.rows() )
        skip_this= next(row_iter)
        return row_iter

class TestEmbeddedSchemaSheet( unittest.TestCase ):
    def setUp( self ):
        self.wb= SheetMockWorkbook( )
        self.sheet= stingray.sheet.EmbeddedSchemaSheet( self.wb, ’The_Name’, loader_class=MockLoader
        self.wb.rows = [
            MockRow( self.sheet, stingray.cell.TextCell("Col 1", self.wb),
              stingray.cell.TextCell("Col 2", self.wb) ),
            MockRow( self.sheet, stingray.cell.NumberCell(1, self.wb),
              stingray.cell.TextCell("First", self.wb) ),
            MockRow( self.sheet, stingray.cell.NumberCell(2, self.wb),
              stingray.cell.TextCell("Second", self.wb), ),
            ]
    def test_should_iterate( self ):
        rows_as_dict_iter = (
            dict( zip( (a.name for a in self.sheet.schema), r ) )
            for r in self.sheet.rows() )
        row_list= list(  rows_as_dict_iter )
        self.assertEqual( ’The_Name’, self.wb.requested )
        self.assertEqual( 2, len(row_list) )
        self.assertTrue( all( isinstance(r,dict) for r in row_list ) )
        row= row_list[0]
        self.assertEqual( 1, row[’Col 1’].to_int() )
        self.assertEqual( "First", row[’Col 2’].to_str() )
        row= row_list[1]
        self.assertEqual( 2, row[’Col 1’].to_int() )
        self.assertEqual( "Second", row[’Col 2’].to_str() )
    def test_should_have_attributes( self ):
        self.assertEqual( ’The_Name’, self.sheet.name )
```

Test Suite and Runner

In case we want to build up a larger test suite, we avoid doing any real work unless this is the main module being executed.

```
import test
suite= test.suite_maker( globals() )

if __name__ == "__main__":
    print( __file__ )
    unittest.TextTestRunner().run(suite())
```

1.14.6 Workbook Testing

There are many types of workbooks. See *Workbook Package – Uniform Wrappers for Workbooks* for details.

These tests use actual files in the `sample` directory.

- `csv_workbook.csv` used by CSV_Workbook Tests.
- `excel97_workbook.xls` used by XLS_Workbook Tests.
- `excel_workbook.xlsx` used by XLSX_Workbook Tests.
- `ooo_workbook.ods` used by ODS_Workbook Tests.
- `tab_workbook.tab` used by CSV_Workbook with Tab delimiters.
- `workbook.simple` used by Fixed_Workbook Tests. `simple.csv` is the schema.
- `numbers_workbook_09.numbers` used by Numbers09_Workbook Tests.
- `numbers_workbook_13.numbers` used by Numbers13_Workbook Tests.

Overheads

```
"""stingray.workbook Unit Tests."""
import unittest
import os
import decimal
import datetime
import stingray.sheet
import stingray.schema
import stingray.schema.loader
import stingray.workbook
```

CSV_Workbook Tests

A `workbook.Workbook` produces a list of sheet names, is a factory for individual `Sheet` instances and produces the rows of a named sheet. Also, a `workbook.Workbook` is a context.

All subclasses of `workbook.Workbook` must be polymorphic with this degenerate special cases. This means that the tests are nearly identical.

The most important distinction among the various physical formats is that other formats (XLS, XLSX, ODS, etc.) support multiple sheets in a single workbook, where CSV (and Fixed) have only a single sheet in the workbook.

```
class TestCSVWorkbook( unittest.TestCase ):
    theClass = stingray.workbook.CSV_Workbook
    extraKW= {}
    theFile = os.path.join('sample', 'csv_workbook.csv')
    theName = "csv_workbook"
    theSheets = ["csv_workbook"]
    def setUp( self ):
        self.wb = self.theClass( self.theFile, **self.extraKW )
    def test_should_have_sheets( self ):
        self.assertEqual( set(self.theSheets), set(self.wb.sheets()) )
    def test_should_open_sheet_1( self ):
        s = self.wb.sheet( 'Sheet1' )
        row_list= list( s.rows() )
        self.assertEqual( 3, len(row_list) )
        row = row_list[0]
        self.assertEqual( 7, len(row) )
        self.assertEqual( "Col 1 - int", row[0].to_str() )
        self.assertEqual( "Col 2.0 - float", row[1].to_str() )
        self.assertEqual( 'Column "3" - string', row[2].to_str() )
        self.assertEqual( "Column '4' - date", row[3].to_str() )
        self.assertEqual( "Column 5 - boolean", row[4].to_str() )
        self.assertEqual( "Column 6 - empty", row[5].to_str() )
        self.assertEqual( "Column 7 - Error", row[6].to_str() )
    def test_should_have_data_sheet_1( self ):
        s = self.wb.sheet( 'Sheet1' )
        row_list= list( s.rows() )
        self.assertEqual( 3, len(row_list) )
        row = row_list[1]
        self.assertEqual( 42, row[0].to_int() )
        self.assertAlmostEqual( 3.1415926, row[1].to_float() )
        self.assertEqual( 'string', row[2].to_str() )
    def test_should_create_context( self ):
        with self.theClass( self.theFile ) as ctx:
            self.assertEqual( set(self.theSheets), set(ctx.sheets()) )
    def test_should_use_context( self ):
        with self.wb:
            self.assertEqual( set(self.theSheets), set(self.wb.sheets()) )
```

CSV_Workbook with Tab delimiters

The idea is that we have a very small change to handle different delimiters in the `csv` module.

```
class TestTabWorkbook( unittest.TestCase ):
    theClass = stingray.workbook.CSV_Workbook
    extraKW = { ’delimiter’:’\t’ }
    theFile = os.path.join(’sample’, ’tab_workbook.tab’)
    theName = "tab_workbook"
    theSheets = ["tab_workbook"]
```

Fixed_Workbook Tests

A `workbook.Fixed_Workbook` is similar to a `workbook.CSV_Workbook`. It always has an external schema and the column contents might be different due to encoding or truncation issues.

```
class TestFixedWorkbook( TestCSVWorkbook ):
    theClass = stingray.workbook.Fixed_Workbook
    theFile = os.path.join(’sample’, ’workbook.simple’)
    theName = "workbook"
    theSheets = ["workbook"]
    def setUp( self ):
        schema_wb= stingray.workbook.CSV_Workbook( os.path.join( ’sample’, ’simple.csv’ ) )
        esl = stingray.schema.loader.ExternalSchemaLoader( schema_wb,
            ’sheet1’ )
        self.schema= esl.schema()
        self.wb = self.theClass( self.theFile, schema=self.schema )
    def test_should_open_sheet_1( self ):
        s = self.wb.sheet( ’Sheet1’ )
        row_list= list( s.rows() )
        self.assertEqual( 3, len(row_list) )
        row = row_list[0]
        self.assertEqual( 7, len(row) )
        self.assertEqual( "Col 1 - int", row[0].to_str() )
        self.assertEqual( "Col 2.0 - f", row[1].to_str() )
        self.assertEqual( ’Column "3"’, row[2].to_str() )
        self.assertEqual( "Column ’4’", row[3].to_str() )
        self.assertEqual( "Column 5 -", row[4].to_str() )
        self.assertEqual( "Column 6 -", row[5].to_str() )
        self.assertEqual( "Column 7 -", row[6].to_str() )
    def test_should_create_context( self ):
        with self.theClass( self.theFile, schema=self.schema ) as ctx:
            self.assertEqual( set(self.theSheets), set(ctx.sheets()) )
```

ODS_Workbook Tests

An ODS workbook is the gold standard.

```
class TestODSWorkbook( TestCSVWorkbook ):
    theClass = stingray.workbook.ODS_Workbook
    theFile = os.path.join(’sample’, ’ooo_workbook.ods’)
    theName = "ooo_workbook"
    theSheets = ['Sheet1','Sheet2','Sheet3']
    def test_should_open_sheet_2( self ):
        s = self.wb.sheet( 'Sheet2',
            stingray.sheet.EmbeddedSchemaSheet,
            loader_class= stingray.schema.loader.HeadingRowSchemaLoader )
        row_list= list( s.rows() )
        #print( 'row_list=', row_list )
        self.assertEqual( 3, len(row_list) )
        # First row (after the heading)
        row = row_list[0]
        self.assertEqual( 1, row[0].to_int() )
        self.assertEqual( 'data', row[1].to_str() )
        # Headings
        self.assertEqual( 2, len(s.schema) )
        self.assertEqual( 'Sheet 2 \u2013 int', s.schema[0].name )
        self.assertEqual( 'Sheet 2 \u2013 string', s.schema[1].name )
```

XLS_Workbook Tests

An XLS workbook can have multiple sheets. Each sheet can have a different schema. This should appear to be like an ODS workbook.

```
class TestXLSWorkbook( TestODSWorkbook ):
    theClass = stingray.workbook.XLS_Workbook
    theFile = os.path.join('sample', 'excel97_workbook.xls')
    theName = "excel97_workbook"
    theSheets = ['Sheet1','Sheet2','Sheet3']
```

XLSX_Workbook Tests

An XLSX workbook should be functionally identical to an ODS workbook. The physical format is remarkably different, but the content must appear identical.

```
class TestXLSXWorkbook( TestODSWorkbook ):
    theClass = stingray.workbook.XLSX_Workbook
    theFile = os.path.join('sample', 'excel_workbook.xlsx')
    theName = "excel_workbook"
    theSheets = ['Sheet1','Sheet2','Sheet3']
```

Numbers09_Workbook Tests

A Numbers '09 workbook should be functionally similar to an ODS workbook. Numbers has Workspaces (a/k/a "Pages") with Tables. The list of "sheets" is (workspace,table) pairs.

```
class TestNumbers09Workbook( TestCSVWorkbook ):
    theClass = stingray.workbook.Numbers09_Workbook
    theFile = os.path.join('sample', 'numbers_workbook_09.numbers')
    theName = "numbers_workbook_09"
    theSheets = [('Sheet 1','Table 1'), ('Sheet 2','Table 1'), ('Sheet 3', 'Table 1')]
    def test_should_open_sheet_1( self ):
        s = self.wb.sheet( ('Sheet 1','Table 1') )
        row_list= list( s.rows() )
        self.assertEqual( 3, len(row_list) )
        row = row_list[0]
        self.assertEqual( 7, len(row) )
        self.assertEqual( "Col 1 - int", row[0].to_str() )
        self.assertEqual( "Col 2.0 - float", row[1].to_str() )
        self.assertEqual( 'Column "3" - string', row[2].to_str() )
        self.assertEqual( "Column '4' - date", row[3].to_str() )
        self.assertEqual( "Column 5 - boolean", row[4].to_str() )
        self.assertEqual( "Column 6 - empty", row[5].to_str() )
        self.assertEqual( "Column 7 - Error", row[6].to_str() )
    def test_should_read_workbook( self ):
        """Read from workbook instead of sheet."""
        s = self.wb.sheet( ('Sheet 1','Table 1') )
        row_list_wb = list( self.wb.rows_of( s ) )
        row_list_s= list( s.rows() )
        self.assertEqual( row_list_wb, row_list_s )
    def test_should_have_data_sheet_1( self ):
        s = self.wb.sheet( ('Sheet 1','Table 1') )
        row_list= list( s.rows() )
        self.assertEqual( 3, len(row_list) )
        row = row_list[1]
        self.assertEqual( 42, row[0].to_int() )
        #print( repr(row[1]) )
        self.assertAlmostEqual( decimal.Decimal('3.1415926'), row[1].to_decimal(7) )
        self.assertAlmostEqual( 3.1415926, row[1].to_float() ) # ?
        self.assertEqual( 'string', row[2].to_str() )
        self.assertEqual( datetime.datetime(1956, 9, 10, 0, 0), row[3].to_datetime() )
```

Numbers13_Workbook Tests

A Numbers '13 workbook should be functionally similar to an Numbers '09 workbook. The format, however, is not XML-based. The content must appear as close to identical as practical. Numbers has Workspaces (a/k/a "Pages") with Tables. The list of "sheets" is (workspace,table) pairs.

```
class TestNumbers13Workbook( TestNumbers09Workbook ):
    theClass = stingray.workbook.Numbers13_Workbook
    theFile = os.path.join('sample', 'numbers_workbook_13.numbers')
    theName = "numbers_workbook_13"
    theSheets = [('Sheet 1','Table 1'), ('Sheet 2','Table 1'), ('Sheet 3', 'Table 1')]
```

Workbook Factory Function

The `workbook.open_workbook()` function should provide simple, uniform access to a workbook file.

```
class TestWBFactory( unittest.TestCase ):
    def test_should_open_csv( self ):
        w = stingray.workbook.open_workbook( os.path.join('sample','csv_workbook.csv') )
        self.assertEqual( set(['csv_workbook']), set(w.sheets()) )
    def test_should_open_fixed( self ):
        w = stingray.workbook.open_workbook(
            os.path.join('sample','workbook.simple'),
            stingray.sheet.ExternalSchemaSheet,
            schema_path='sample',
            schema_sheet='simple', )
        self.assertEqual( set(['workbook']), set(w.sheets()) )
    def test_should_open_xls( self ):
        w = stingray.workbook.open_workbook( os.path.join('sample','excel97_workbook.xls') )
        self.assertEqual( set([’Sheet1’,’Sheet2’,’Sheet3’]), set(w.sheets()) )
    def test_should_open_xlsx( self ):
        w = stingray.workbook.open_workbook( os.path.join(’sample’,’excel_workbook.xlsx’) )
        self.assertEqual( set([’Sheet1’,’Sheet2’,’Sheet3’]), set(w.sheets()) )
    def test_should_open_ods( self ):
        w = stingray.workbook.open_workbook( os.path.join(’sample’,’ooo_workbook.ods’) )
        self.assertEqual( set([’Sheet1’,’Sheet2’,’Sheet3’]), set(w.sheets()) )
```

Test Suite and Runner

In case we want to build up a larger test suite, we avoid doing any real work unless this is the main module being executed.

```
import test
suite= test.suite_maker( globals() )

if __name__ == "__main__":
    print( __file__ )
    unittest.TextTestRunner().run(suite())
```

1.14.7 COBOL Schema Testing

Processing COBOL (and possibly EBCDIC) files requires a more sophisticated schema. And a more sophisticated schema loader.

See *The COBOL Package*.

Overheads

```
"""stingray.cobol Unit Tests."""
import unittest
import decimal
import weakref
import logging, sys
import io
import types

import stingray.cobol
```

Handy Mocks

```
class MockDefaultCell:
    def __init__( self, value, workbook, attr ):
        self.value= value
        self.workbook= workbook
        sekf.attr= attr
    def __eq__( self, other ):
        return all( (
            self.__class__ == other.__class__,
            self.value == other.value,
            self.attr == other.attr
            ) )
    def __ne__( self, other ):
        return not self.__eq__( other )

class MockDDE:
    def __init__( self, **kw ):
        self.totalSize= 0 # if otherwise unspecified
        self.kw= kw
        self.__dict__.update( kw )
        self.children=[]
    def __repr__( self ):
        nv= ", ".join( "{0}={1!r}".format( k, self.kw[k] ) for k in self.kw )
        return "MockDDE( {1} )".format( self.children, nv )
    def addChild( self, child ):
        self.children.append( child )
        self.totalSize += child.totalSize

class MockNonRedefine:
    def __init__( self, **kw ):
        self.__dict__.update( kw )

class MockOccurs:
    def __init__( self, number ):
        self._number= number
    def number( self, aRow ):
        return self._number
    def __repr__( self ):
        return "{_class_name}({_number})".format(_class_name=self.__class__.__name__, **self.__dict__

class MockSchema( list ):
    def __init__( self, *args, **kw ):
        super().__init__( *args )
        self.info= kw
    def lrecl( self ):
        return max( a.offset+a.size for a in self )
```

Repeating Attribute

We subclass `schema.Attribute` to make it depend on `MockDefaultCell` instead of some working class in `cell`.

```
class AttributeFixture( stingray.cobol.RepeatingAttribute ):
    default_cell= MockDefaultCell
```

Now we can test simple DDE's without an occurs clause.

```
class TestNonRepeatingAttribute( unittest.TestCase ):
    def setUp( self ):
        self.simple_dde= MockDDE( level="05", name="SOME-COLUMN", occurs=MockOccurs(1),
            size=5, totalSize=5, offset=0 )
        self.simple_dde.dimensionality= ()
        self.simple= AttributeFixture(
            name="SOME-COLUMN", dde=weakref.ref(self.simple_dde),
            size=5, create=MockDefaultCell,  )
    def test_should_have_names( self ):
        self.assertEqual( "SOME-COLUMN", self.simple.name )
    def test_should_not_index_simple( self ):
        try:
            simple= self.simple.index(2)
            self.fail()
        except IndexError:
            pass
```

And we can test simple DDE's with an occurs clause.

```
class TestRepeatingAttribute( unittest.TestCase ):
    def setUp( self ):
        self.dde= MockDDE( level="05", name="REPEAT", occurs=MockOccurs(4), size=3, totalSize=12, of
        self.dde.dimensionality=(self.dde,)
        self.complex= AttributeFixture(
            name="REPEAT", dde=weakref.ref(self.dde),
            create=MockDefaultCell, occurs=4, size=3, totalSize=12, )
    def test_should_have_names( self ):
        self.assertEqual( "REPEAT", self.complex.name )
    def test_should_index_complex( self ):
        """1-based indexing in COBOL is ignored here."""
        complex= self.complex.index(0)
        self.assertEqual( 5, complex.offset )
        self.assertEqual( 3, complex.size )
        complex= self.complex.index(1)
        self.assertEqual( 8, complex.offset )
        self.assertEqual( 3, complex.size )
        complex= self.complex.index(2)
        self.assertEqual( 11, complex.offset )
        self.assertEqual( 3, complex.size )
        complex= self.complex.index(3) # out of range!
        self.assertEqual( 14, complex.offset )
        self.assertEqual( 3, complex.size )
```

And we can test nested DDE's with multiple occurs clauses.

```
class TestNestedRepeatingAttribute( unittest.TestCase ):
    def setUp( self ):
        self.child= MockDDE( level="10", name="ITEM",
            occurs=MockOccurs(4), size=2, totalSize=8, offset=5,
            )
        self.parent= MockDDE( level="05", name="REPEAT",
            occurs=MockOccurs(5), size=8, totalSize=30,  offset=5, children=[self.child],
            )
        self.parent.dimensionality=(self.parent,)
        self.child.dimensionality=(self.parent,self.child)
        self.complex_05= AttributeFixture(
            name="REPEAT", dde=weakref.ref(self.parent),
            create=MockDefaultCell, occurs=5, size=8, totalSize=40, )
        self.complex_10= AttributeFixture(
            name="ITEM", dde=weakref.ref(self.child),
            create=MockDefaultCell, occurs=4, size=2, totalSize=8,  )
    def test_should_have_names( self ):
        self.assertEqual( "REPEAT", self.complex_05.name )
        self.assertEqual( "ITEM", self.complex_10.name )
    def test_should_index_complex_item( self ):
        """Partial index works from top to bottom.
        COBOL is 1-based, but that's ignored here."""
        complex= self.complex_10.index(0)
        self.assertEqual( 5, complex.offset )
        self.assertEqual( 2, complex.size )
        complex= self.complex_10.index(1)
        self.assertEqual( 13, complex.offset )
        self.assertEqual( 2, complex.size )
        complex= self.complex_10.index(2)
        self.assertEqual( 21, complex.offset )
        self.assertEqual( 2, complex.size )
        complex= self.complex_10.index(3)
        self.assertEqual( 29, complex.offset )
        self.assertEqual( 2, complex.size )
    def test_should_index_complex_group( self ):
        complex= self.complex_10.index(1,0)
        self.assertEqual( 13, complex.offset )
        self.assertEqual( 2, complex.size )
        complex= self.complex_10.index(1,1)
        self.assertEqual( 15, complex.offset )
        self.assertEqual( 2, complex.size )
        complex= self.complex_10.index(2,0)
        self.assertEqual( 21, complex.offset )
        self.assertEqual( 2, complex.size )
        complex= self.complex_10.index(2,1)
        self.assertEqual( 23, complex.offset )
        self.assertEqual( 2, complex.size )
```

DDE Data Access

The data access mediated by a DDE-as-schema is a bit more complex than the trivial (flat) data access mediated by the schema representation from the schema package.

Note that we don’t test sheet.LazyRow on workbook.Fixed_Workbook. It doesn’t work properly because the workbook.Fixed_Workbook doesn’t know about the new cobol.RepeatingAttribute that models repeating groups.

The cobol.Character_File, however, can depend on cobol.RepeatingAttribute.

```
class Test_Dimensional_Access( unittest.TestCase ):
    def setUp( self ):
        self.top= MockDDE( level=’01’, name=’SURVEY-RESPONSES’,
            allocation=MockNonRedefine(), picture=None, offset=0, size=60, occurs=MockOccurs(1),
            variably_located= False,
            )
        self.top.top= self.top
        self.group_05 = MockDDE( level=’05’, name=’QUESTION-NUMBER’,
            allocation=MockNonRedefine(), picture=None, occurs=MockOccurs(10), offset=0, totalSize=60
        self.top.addChild( self.group_05 )
        self.group_10 = MockDDE( level=’10’, name=’RESPONSE-CATEGORY’,
            allocation=MockNonRedefine(), picture=None, occurs=MockOccurs(3), offset=0, totalSize=6,
        self.group_05.addChild( self.group_10 )
        self.group_15 = MockDDE( level=’15’, name=’ANSWER’,
            allocation=MockNonRedefine(), picture="99", offset=0, totalSize=2, size=2 )
        self.group_15.dimensionality= (self.group_05, self.group_10,)
        self.group_10.addChild( self.group_15 )

        self.schema= MockSchema(
            (stingray.cobol.RepeatingAttribute( name="ANSWER", dde=weakref.ref(self.group_15),
                size=2, totalSize=60, ),
            ),
            dde= [self.top]
        )
        self.data = stingray.cobol.Character_File( name="",
            file_object= ["111213212223313233414243515253616263717273818283919293010203",],
            schema=self.schema )

    def test_should_get_cell( self ):
        """Get individual cell values."""
        # 1-based indexing in COBOL, Python, however, is zero-based.
        row= next( self.data.sheet( "" ).rows() )
        self.assertEqual( "11", row.cell( self.schema[0].index(0,0) ).to_str() )
        self.assertEqual( "21", row.cell( self.schema[0].index(1,0) ).to_str() )
        self.assertEqual( "22", row.cell( self.schema[0].index(1,1) ).to_str() )
        self.assertEqual( "23", row.cell( self.schema[0].index(1,2) ).to_str() )
        self.assertEqual( "93", row.cell( self.schema[0].index(8,2) ).to_str() )
    def test_should_get_array( self ):
        """Get tuple of values from incomplete index specification."""
        # 1-based indexing in COBOL, Python, however, is zero-based.
        row= next( self.data.sheet( "" ).rows() )
        #print( "Original and Tweaked =", self.schema[0], self.schema[0].index(1) )
        slice= row.cell( self.schema[0].index(1) )
        #print( "Offset, Slice =", self.schema[0].index(1).offset, slice )
        self.assertEqual( ('21', '22', '23'), tuple( c.to_str() for c in slice ) )
```

Workbook File Access

Character File and EBCDIC File access covers most of the bases.

Here are mocks to define a schema and the resulting Cell instances.

```
class MockAttribute:
    def __init__( self, **kw ):
        self.__dict__.update( kw )
    def __repr__( self ):
        return repr( self.__dict__ )

class MockTextCell:
    def __init__( self, buffer, workbook, attr ):
        self.raw, self.workbook, self.attr= buffer, workbook, attr
        self.value= workbook.text( self.raw, self.attr )

class MockDisplayCell( MockTextCell ):
    def __init__( self, buffer, workbook, attr ):
        self.raw, self.workbook, self.attr= buffer, workbook, attr
        self.value= workbook.number_display( self.raw, self.attr )

class MockComp3Cell( MockTextCell ):
    def __init__( self, buffer, workbook, attr ):
        self.raw, self.workbook, self.attr= buffer, workbook, attr
        self.value= workbook.number_comp3( self.raw, self.attr )

class MockCompCell( MockTextCell ):
    def __init__( self, buffer, workbook, attr ):
        self.raw, self.workbook, self.attr= buffer, workbook, attr
        self.value= workbook.number_comp( self.raw, self.attr )
```

In principle, we need to mock the `sheet.ExternalSchemaSheet` that gets built.

Given a schema and some data, pick apart the row of a Character File. Generally, we'll deal with ordinary text using no particular encoding. Python can generally handle ASCII bytes or Unicode pretty simply via the "encoding" argument to the open().

```
class TestCharacterFile_Text( unittest.TestCase ):
    def setUp( self ):
        self.parent= MockDDE( level="01", name="PARENT", variably_located= False, )
        self.attr_word= MockAttribute(
            name="WORD", offset=0, size=3, create=MockTextCell, dimensionality=None,
            size_scale_precision=("XXX",True,3,0,0,False,None), )
        self.attr_num1= MockAttribute(
            name="NUMBER-1", offset=3, size=6, create=MockDisplayCell, picture="999.99", dimensionali
            size_scale_precision=("999.99",False,6,0,2,True,"."), )
        self.attr_num2= MockAttribute(
            name="NUMBER-2", offset=9, size=5, create=MockDisplayCell, picture="S999V99", dimensional
            size_scale_precision=("999V99",False,6,0,2,True,"V"), )
        self.schema= MockSchema(
            ( self.attr_word, self.attr_num1, self.attr_num2
            ),
            dde= [ self.parent ],
        )
        self.data= ["ASC123.4567890XYZ",]
        self.wb= stingray.cobol.Character_File( "name", self.data, self.schema )
    def test_should_get_cells( self ):
        row= next( self.wb.sheet( "IGNORED" ).rows() )
        self.assertEqual( "ASC", row.cell( self.attr_word ).value )
        self.assertEqual( decimal.Decimal(’123.45’), row.cell( self.attr_num1 ).value )
        self.assertEqual( decimal.Decimal(’678.90’), row.cell( self.attr_num2 ).value )
```

Do we need TestCharacterFile_Bytes as a separate test case? Probably not. The cobol.Character_File is a workbook.Fixed_Workbook which is text, not bytes.

Given a schema and some data, pick apart the row of a fake EBCDIC File. This file has the default RECFM of F – no blocking and no header words.

```
class TestEBCDICFile_Fixed( unittest.TestCase ):
    def setUp( self ):
        self.parent= MockDDE( level="01", name="PARENT", variably_located= False, )
        self.attr_word= MockAttribute(
            name="WORD", offset=0, size=3, create=MockTextCell, dimensionality=None,
            size_scale_precision=("XXX",True,3,0,0,False,None), )
        self.attr_num1= MockAttribute(
            name="NUMBER-1", offset=3, size=6, create=MockDisplayCell, picture="999.99", dimensionali
            size_scale_precision=("999.99",False,6,0,2,True,"."), )
        self.attr_num2= MockAttribute(
            name="NUMBER-2", offset=9, size=5, create=MockDisplayCell, picture="S999V99", dimensional
            size_scale_precision=("999V99",False,5,0,2,True,"V"), )
        self.attr_comp= MockAttribute(
            name="NUMBER-3", offset=14, size=4, create=MockCompCell, picture="S99999999", dimensional
            size_scale_precision=("999V99",False,5,0,2,True,"V"), )
        self.attr_comp3= MockAttribute(
            name="NUMBER-4", offset=18, size=3, create=MockComp3Cell, picture="S999V99", dimensionali
            size_scale_precision=("999V99",False,3,0,2,True,"V"), )
        self.schema= MockSchema(
            ( self.attr_word, self.attr_num1, self.attr_num2, self.attr_comp, self.attr_comp3
            ),
            dde= [ self.parent ],
        )
    def test_should_get_cells( self ):
        self.data= io.BytesIO(
            b"\xe9\xd6\xe2" # WORD="ZOS"
            b"\xf1\xf2\xf3K\xf4\xf5" # NUMBER-1="123.45"
            b"\xf6\xf7\xf8\xf9\xf0" # NUMBER-2="678.90"
            b"\x00\x00\x12\x34" # NUMBER-3=4660
            b"\x98\x76\x5d" # NUMBER-4=-987.65
        )
        self.wb= stingray.cobol.EBCDIC_File( "name", self.data, self.schema )
        row= next( self.wb.sheet( "IGNORED" ).rows() )
        self.assertEqual( 21, self.schema.lrecl() ) # last field has offset 18, size 3
        self.assertEqual( "ZOS", row.cell( self.attr_word ).value )
        self.assertEqual( decimal.Decimal(’123.45’), row.cell( self.attr_num1 ).value )
        self.assertEqual( decimal.Decimal(’678.90’), row.cell( self.attr_num2 ).value )
        self.assertEqual( decimal.Decimal(’4660’), row.cell( self.attr_comp ).value )
        self.assertEqual( decimal.Decimal(’-987.65’), row.cell( self.attr_comp3 ).value )
```

Given a schema and some data, pick apart the row of a fake EBCDIC File. This file has a RECFM of V, it has Record Descriptor Word (RDW)

```
class TestEBCDICFile_Variable( TestEBCDICFile_Fixed ):
    def test_should_get_cells( self ):
        """Data has 4 byte RDW in front of the row."""
        self.data= io.BytesIO(
            b"\x00\x19\x00\x00" # RDW
            b"\xe9\xd6\xe2" # WORD="ZOS"
            b"\xf1\xf2\xf3K\xf4\xf5" # NUMBER-1="123.45"
            b"\xf6\xf7\xf8\xf9\xf0" # NUMBER-2="678.90"
            b"\x00\x00\x12\x34" # NUMBER-3=4660
            b"\x98\x76\x5d" # NUMBER-4=-987.65
        )
        self.wb= stingray.cobol.EBCDIC_File( "name", self.data, self.schema, RECFM="V" )
        row= next( self.wb.sheet( "IGNORED" ).rows() )
        self.assertEqual( 21, self.schema.lrecl() ) # last field has offset 18, size 3
        self.assertEqual( "ZOS", row.cell( self.attr_word ).value )
        self.assertEqual( decimal.Decimal(’123.45’), row.cell( self.attr_num1 ).value )
        self.assertEqual( decimal.Decimal(’678.90’), row.cell( self.attr_num2 ).value )
        self.assertEqual( decimal.Decimal(’4660’), row.cell( self.attr_comp ).value )
        self.assertEqual( decimal.Decimal(’-987.65’), row.cell( self.attr_comp3 ).value )
```

EBCDIC File with VB format. It has Block Descriptor Word (BDW) and Record Descriptor Word (RDW).

```
class TestEBCDICFile_VariableBlocked( TestEBCDICFile_Fixed ):
    def test_should_get_cells( self ):
        """Data has 4 byte BDW and 4 byte RDW in front of the row."""
        # Build 2 blocks.
        self.data= io.BytesIO(
        b"\x00\x1d\x00\x00" #BDW
            b"\x00\x19\x00\x00" # RDW
                b"\xe9\xd6\xe2" # WORD="ZOS"
                b"\xf1\xf2\xf3K\xf4\xf5" # NUMBER-1="123.45"
                b"\xf6\xf7\xf8\xf9\xf0" # NUMBER-2="678.90"
                b"\x00\x00\x12\x34" # NUMBER-3=4660
                b"\x98\x76\x5d" # NUMBER-4=-987.65
        b"\x00\x1d\x00\x00" #BDW
            b"\x00\x19\x00\x00" # RDW
                b"\xe9\xd6\xe2" # WORD="ZOS"
                b"\xf1\xf2\xf3K\xf4\xf5" # NUMBER-1="123.45"
                b"\xf6\xf7\xf8\xf9\xf0" # NUMBER-2="678.90"
                b"\x00\x00\x12\x34" # NUMBER-3=4660
                b"\x98\x76\x5d" # NUMBER-4=-987.65
        )
        self.wb= stingray.cobol.EBCDIC_File( "name", self.data, self.schema, RECFM="VB" )
        row_iter= self.wb.sheet( "IGNORED" ).rows()
        row= next( row_iter )
        self.assertEqual( 21, self.schema.lrecl() ) # last field has offset 18, size 3
        self.assertEqual( "ZOS", row.cell( self.attr_word ).value )
        self.assertEqual( decimal.Decimal('123.45'), row.cell( self.attr_num1 ).value )
        self.assertEqual( decimal.Decimal('678.90'), row.cell( self.attr_num2 ).value )
        self.assertEqual( decimal.Decimal('4660'), row.cell( self.attr_comp ).value )
        self.assertEqual( decimal.Decimal('-987.65'), row.cell( self.attr_comp3 ).value )
        row2= next( row_iter )
        with self.assertRaises(StopIteration):
            row3= next( row_iter )
```

Given a schema and some data, pick apart the row of a fake EBCDIC File. This file has a RECFM of V, but not Record Descriptor Word (RDW). The RECFM=N should be able to parse it, however.

```
class TestEBCDICFile_VariableWithoutRDW( TestEBCDICFile_Fixed ):
    def test_should_get_cells( self ):
        """Data lacks a 4 byte RDW in front of the row."""
        buffer= (
            b"\xe9\xd6\xe2" # WORD="ZOS"
            b"\xf1\xf2\xf3K\xf4\xf5" # NUMBER-1="123.45"
            b"\xf6\xf7\xf8\xf9\xf0" # NUMBER-2="678.90"
            b"\x00\x00\x12\x34" # NUMBER-3=4660
            b"\x98\x76\x5d" # NUMBER-4=-987.65

            b"\xe9\xd6\xe2" # WORD="ZOS"
            b"\xf1\xf2\xf3K\xf4\xf5" # NUMBER-1="123.45"
            b"\xf6\xf7\xf8\xf9\xf0" # NUMBER-2="678.90"
            b"\x00\x00\x12\x34" # NUMBER-3=4660
            b"\x98\x76\x5d" # NUMBER-4=-987.65
        )
        self.data= io.BytesIO( buffer )
        self.wb= stingray.cobol.EBCDIC_File( "name", self.data, self.schema, RECFM="N" )
        row_iter= self.wb.sheet( "IGNORED" ).rows()
        row= next( row_iter )
        self.assertEqual( 21, self.schema.lrecl() ) # last field has offset 18, size 3
        self.assertEqual( "ZOS", row.cell( self.attr_word ).value )
        self.assertEqual( decimal.Decimal('123.45'), row.cell( self.attr_num1 ).value )
        self.assertEqual( decimal.Decimal('678.90'), row.cell( self.attr_num2 ).value )
        self.assertEqual( decimal.Decimal('4660'), row.cell( self.attr_comp ).value )
        self.assertEqual( decimal.Decimal('-987.65'), row.cell( self.attr_comp3 ).value )
        row2= next( row_iter )
        with self.assertRaises(StopIteration):
            row3= next( row_iter )
```

Todo

EBCDIC File V format with Occurs Depending On to show the combination.

EBCDIC file with bad numeric data. This doesn't use the above Mocks. It uses real Cell definitions and real Usage definitions because the real Usage definitions tolerate bad data.

```
class TestEBCDICFile_Invalid( unittest.TestCase ):
    def setUp( self ):
        self.parent= MockDDE( level="01", name="PARENT", variably_located= False, )
        self.display= stingray.cobol.defs.UsageDisplay("DISPLAY")
        self.display.numeric= True
        self.comp= stingray.cobol.defs.UsageComp("COMP")
        self.comp3= stingray.cobol.defs.UsageComp3("COMP-3")
        self.attr_word= MockAttribute(
            name="WORD", offset=0, size=3, create=stingray.cobol.defs.TextCell, dimensionality=None,
            size_scale_precision=("XXX",True,3,0,0,False,None), )
        self.attr_num1= MockAttribute(
            name="NUMBER-1", offset=3, size=6, create=self.display.create_func, picture="999.99", dir
            size_scale_precision=("999.99",False,6,0,2,True,"."), )
        self.attr_num2= MockAttribute(
            name="NUMBER-2", offset=9, size=5, create=self.display.create_func, picture="S999V99", di
            size_scale_precision=("999V99",False,6,0,2,True,"V"), )
        self.attr_comp= MockAttribute(
            name="NUMBER-3", offset=14, size=4, create=self.comp.create_func, picture="S99999999", di
            size_scale_precision=("999V99",False,6,0,2,True,"V"), )
        self.attr_comp3= MockAttribute(
            name="NUMBER-4", offset=18, size=5, create=self.comp3.create_func, picture="S999V99", dir
            size_scale_precision=("999V99",False,6,0,2,True,"V"), )
        self.schema= MockSchema(
            ( self.attr_word, self.attr_num1, self.attr_num2, self.attr_comp, self.attr_comp3
            ),
            dde= [ self.parent ],
        )
    def test_should_get_error_cells( self ):
        self.data= io.BytesIO( b"\xe9\xd6\xe2\xf1\xd6\xf3K\xf4\xd6\xf6\xf7\xf8\xd6\xf0\x00\x00\x12\x
        self.wb= stingray.cobol.EBCDIC_File( "name", self.data, self.schema )
        row= next( self.wb.sheet( "IGNORED" ).rows() )
        self.assertEqual( 23, self.schema.lrecl() )
        self.assertEqual( "ZOS", row.cell( self.attr_word ).value )
        self.assertIsNone( row.cell( self.attr_num1 ).value )
        self.assertEqual( b'\xf1\xd6\xf3K\xf4\xd6', row.cell( self.attr_num1 ).raw )
        self.assertIsNone( row.cell( self.attr_num2 ).value )
        self.assertEqual( b'\xf6\xf7\xf8\xd6\xf0', row.cell( self.attr_num2 ).raw )
        # Should Work: can't easily corrupt USAGE COMPUTATIONAL data.
        self.assertEqual( decimal.Decimal('4822'), row.cell( self.attr_comp ).value )
        self.assertEqual( b'\x00\x00\x12\xd6', row.cell( self.attr_comp ).raw )
        # Should this work? Not clear.
        #self.assertIsNone( row.cell( self.attr_comp3 ).value )
        self.assertEqual( decimal.Decimal('0'), row.cell( self.attr_comp3 ).value )
        self.assertEqual( b'\x00\x00\x00\x00\x00', row.cell( self.attr_comp3 ).raw )
```

Test Conversions

The various PIC and USAGE format subtleties lead to a number of conversion test cases.

```
class TestNumberConversion( unittest.TestCase ):
    def setUp( self ):
        self.wb= stingray.cobol.EBCDIC_File
    def test_should_convert_display(self):
        attr1= types.SimpleNamespace(
            size_scale_precision = ('999v99', False, 5, 0, 2, True, "V") )
        n1= self.wb.number_display( b'\xf1\xf2\xf3\xf4\xf5', attr1 )
        self.assertEqual( decimal.Decimal('123.45'), n1 )
        attr2= types.SimpleNamespace(
            size_scale_precision = ('99999V', False, 5, 0, 0, True, "V") )
        n2= self.wb.number_display( b'\xf1\xf2\xf3\xf4\xf5', attr2 )
        self.assertEqual( decimal.Decimal('12345'), n2 )
        attr3= types.SimpleNamespace(
            size_scale_precision = ('999.99', False, 5, 0, 2, True, ".") )
        n3= self.wb.number_display( b'\xf1\xf2\xf3K\xf4\xf5', attr3 )
        self.assertEqual( decimal.Decimal('123.45'), n3 )
    def test_should_convert_comp3(self):
        attr1= types.SimpleNamespace(
            size_scale_precision = ('999v99', False, 5, 0, 2, True, "V") )
        n1= self.wb.number_comp3( b'\x12\x34\x98\x76\x5d', attr1 )
        self.assertEqual( decimal.Decimal('-1234987.65'), n1 )
        attr2= types.SimpleNamespace(
            size_scale_precision = ('99999V', False, 5, 0, 0, True, "V") )
        n2= self.wb.number_comp3( b'\x12\x34\x98\x76\x5d', attr2 )
        self.assertEqual( decimal.Decimal('-123498765'), n2 )
```

Test Suite and Runner

In case we want to build up a larger test suite, we avoid doing any real work unless this is the main module being executed.

```
import test
suite= test.suite_maker( globals() )

if __name__ == "__main__":
    with test.Logger( stream=sys.stdout, level=logging.DEBUG ):
        logging.info( __file__ )
        unittest.TextTestRunner().run(suite())
        #unittest.main( Test_Dimensional_Access() ) # Specific debugging
```

1.14.8 COBOL Schema Loader Unit Tests

These are unit tests of various parts of the COBOL schema loader. See *COBOL Loader Module – Parse COBOL Source to Load a Schema*.

Overheads

```
"""stingray.cobol.loader Unit Tests."""
import unittest
import stingray.cobol.loader
import weakref
```

Lexical Scanner

At a low-level, a parser requires a lexical scanner to break the input into discrete tokens. In the case of COBOL, there are mercifully few rules to follow.

A few test cases will demonstrate that line-ending `.` as well as a simple version of COBOL quoting rules are followed. The full COBOL language isn't being parsed, just the part of the language used for DDE's.

```
class TestLexicalScanner( unittest.TestCase ):
    def test_should_scan( self ):
        copy1= """\
      * COPY1.COB
       01  DETAIL-LINE.
           05                              PIC X(7).
           05  QUESTION                    PIC ZZ.
           05                              PIC X(6).
           05  PRINT-YES                   PIC ZZ.
           05                              PIC X(3).
           05  PRINT-NO                    PIC ZZ.
           05                              PIC X(6).
           05  NOT-SURE                    PIC ZZ.
           05                              PIC X(7).
"""
        result= [’01’, ’DETAIL-LINE’, ’.’,
        ’05’, ’PIC’, ’X(7)’, ’.’,
        ’05’, ’QUESTION’, ’PIC’, ’ZZ’, ’.’,
        ’05’, ’PIC’, ’X(6)’, ’.’,
        ’05’, ’PRINT-YES’, ’PIC’, ’ZZ’, ’.’,
        ’05’, ’PIC’, ’X(3)’, ’.’,
        ’05’, ’PRINT-NO’, ’PIC’, ’ZZ’, ’.’,
        ’05’, ’PIC’, ’X(6)’, ’.’,
        ’05’, ’NOT-SURE’, ’PIC’, ’ZZ’, ’.’,
        ’05’, ’PIC’, ’X(7)’, ’.’]
        scanner= stingray.cobol.loader.Lexer().scan( copy1 )
        tokens = list( scanner )
        self.assertEqual( len(result), len(tokens) )
        self.assertEqual( result, tokens )

    def test_should_scan_pic( self ):
        copy2= """\
      * COPY2.COB
       01  DETAIL-LINE.
           05  FANCY-FORMAT                PIC 9(7).99.
           05  SSN-FORMAT                  PIC 999-99-9999.
"""
        result= [’01’, ’DETAIL-LINE’, ’.’,
        ’05’, ’FANCY-FORMAT’, ’PIC’, ’9(7).99’, ’.’,
        ’05’, ’SSN-FORMAT’, ’PIC’, ’999-99-9999’, ’.’]
        scanner= stingray.cobol.loader.Lexer().scan( copy2 )
        tokens = list( scanner )
        self.assertEqual( len(result), len(tokens) )
        self.assertEqual( result, tokens )

    def test_should_scan_quotes( self ):
        copy3= """\
      * COPY3.COB
       01  SIMPLE-LINE.
           05  VALUE-1                PIC X(10) VALUE ’STRING    ’.
           05  VALUE-2                PIC X(10) VALUE "STRING    ".
"""
        result= [ ’01’, ’SIMPLE-LINE’, ’.’,
        ’05’, ’VALUE-1’, ’PIC’, ’X(10)’, ’VALUE’, "’STRING    ’", ’.’,
        ’05’, ’VALUE-2’, ’PIC’, ’X(10)’, ’VALUE’, ’"STRING    "’, ’.’]
        scanner= stingray.cobol.loader.Lexer().scan( copy3 )
        tokens = list( scanner )
        self.assertEqual( len(result), len(tokens) )
        self.assertEqual( result, tokens )

    def test_should_replace_text( self ):
        copy4= """\
      * COPY4.COB
       01  ’XY’-SIMPLE-LINE.
```

```
           05  'XY'-VALUE-1                PIC X(10) VALUE 'STRING    '.
           05  'XY'-VALUE-2                PIC X(10) VALUE "STRING    ".
"""
        result= [ '01', 'AB-SIMPLE-LINE', '.',
        '05', 'AB-VALUE-1', 'PIC', 'X(10)', 'VALUE', "'STRING    '", '.',
        '05', 'AB-VALUE-2', 'PIC', 'X(10)', 'VALUE', '"STRING    "', '.']
        scanner= stingray.cobol.loader.Lexer(replacing=[("'XY'","AB")]).scan( copy4 )
        tokens = list( scanner )
        self.assertEqual( len(result), len(tokens) )
        self.assertEqual( result, tokens )
```

Long Lexical Scanner

Some copybooks have junk at the left and right on each line of source. A separate subclass can handle this.

```
class TestLongLexicalScanner( unittest.TestCase ):
    def setUp( self ):
        self.copy1= """\
      **************************************************************
       01  REPORT-TAPE-DETAIL-RECORD.
           02  RDT-REC-CODE-BYTES.                                      00000130
               03  RDT-REC-CODE-KEY              PIC X.                 00000140
        """
        self.scanner= stingray.cobol.loader.Lexer_Long_Lines().scan( self.copy1 )
    def test_should_scan( self ):
        result= ['01', 'REPORT-TAPE-DETAIL-RECORD', '.',
        '02', 'RDT-REC-CODE-BYTES', '.',
        '03', 'RDT-REC-CODE-KEY', 'PIC', 'X', '.',
        ]
        tokens = list( self.scanner )
        self.assertEqual( len(result), len(tokens) )
        self.assertEqual( result, tokens )
```

Picture Parsing

A picture clause has it's own "sub-language" for describing an elementary piece of data. From the picture clause, we extract a number of features.

final final picture with ()'s expanded.

alpha boolean alpha = True, numeric = False.

length len(final)

scale count of "P" positions

precision digits to the right of the decimal point

signed boolean

decimal "." or "V" or None

```
class TestPictureParser( unittest.TestCase ):
    def test_should_expand( self ):
        pic= stingray.cobol.loader.picture_parser( "X(7)" )
        self.assertEqual( "XXXXXXX", pic.final )
        self.assertTrue( pic.alpha )
        self.assertEqual( 7, pic.length )
        self.assertEqual( 0, pic.scale )
        self.assertEqual( 0, pic.precision )
        self.assertFalse( pic.signed )
        self.assertIsNone( pic.decimal )
    def test_should_handle_z( self ):
        pic= stingray.cobol.loader.picture_parser( "ZZ" )
        self.assertEqual( "ZZ", pic.final )
        self.assertFalse( pic.alpha )
        self.assertEqual( 2, pic.length )
        self.assertEqual( 0, pic.scale )
        self.assertEqual( 0, pic.precision )
        self.assertFalse( pic.signed )
        self.assertIsNone( pic.decimal )
    def test_should_handle_9( self ):
        pic= stingray.cobol.loader.picture_parser( "999" )
        self.assertEqual( "999", pic.final )
        self.assertFalse( pic.alpha )
        self.assertEqual( 3, pic.length )
        self.assertEqual( 0, pic.scale )
        self.assertEqual( 0, pic.precision )
        self.assertFalse( pic.signed )
        self.assertIsNone( pic.decimal )
    def test_should_handle_complex( self ):
        pic= stingray.cobol.loader.picture_parser( "9(5)V99" )
        self.assertEqual( "9999999", pic.final )
        self.assertFalse( pic.alpha )
        self.assertEqual( 7, pic.length )
        self.assertEqual( 0, pic.scale )
        self.assertEqual( 2, pic.precision )
        self.assertFalse( pic.signed )
        self.assertEqual( "V", pic.decimal )
    def test_should_handle_signed( self ):
        pic= stingray.cobol.loader.picture_parser( "S9(7)V99" )
        self.assertEqual( "999999999", pic.final )
        self.assertFalse( pic.alpha )
        self.assertEqual( 9, pic.length )
        self.assertEqual( 0, pic.scale )
        self.assertEqual( 2, pic.precision )
        self.assertTrue( pic.signed )
        self.assertEqual( "V", pic.decimal )
    def test_should_handle_db( self ):
        pic= stingray.cobol.loader.picture_parser( "DB9(5).99" )
        self.assertEqual( "DB99999.99", pic.final )
        self.assertFalse( pic.alpha )
        self.assertEqual( 10, pic.length )
        self.assertEqual( 0, pic.scale )
        self.assertEqual( 2, pic.precision )
        self.assertTrue( pic.signed )
        self.assertEqual( ".", pic.decimal )
    def test_should_handle_signed_and_v( self ):
        pic= stingray.cobol.loader.picture_parser( "S9(4)V" )
        self.assertEqual( "9999", pic.final )
        self.assertFalse( pic.alpha )
        self.assertEqual( 4, pic.length )
        self.assertEqual( 0, pic.scale )
        self.assertEqual( 0, pic.precision )
        self.assertTrue( pic.signed )
        self.assertEqual( "V", pic.decimal )
```

Usage

A Usage object is attached to a DDE to explain how to decode the bytes that will be found in the record. There are many cases in COBOL, but we only really care about three: DISPLAY, COMP and COMP-3.

```
class TestUsageDisplay( unittest.TestCase ):
    def setUp( self ):
        self.usage = stingray.cobol.defs.UsageDisplay( "DISPLAY" )
        self.picture= stingray.cobol.loader.Picture( "99999", False, 5, 0, 2, True, "V" )
    def test_should_show_size( self ):
        self.usage.setTypeInfo( self.picture )
        self.assertEqual( 5, self.usage.size() )
        self.assertEqual( "DISPLAY", self.usage.source() )
```

Note the sizing issue for COMP:

Picture Info	Bytes
if 1<=(int+fract)<=4	2
if 5<=(int+fract)<=9	4
if 10<=(int+fract)<=18	8

```
class TestUsageComp( unittest.TestCase ):
    def setUp( self ):
        self.usage = stingray.cobol.defs.UsageComp( "COMP" )
    def test_should_show_size_999( self ):
        self.picture= stingray.cobol.loader.Picture( "999", False, 3, 0, 0, True, None )
        self.usage.setTypeInfo( self.picture )
        self.assertEqual( 2, self.usage.size() )
        self.assertEqual( "COMP", self.usage.source() )
    def test_should_show_size_S9_4( self ):
        self.picture= stingray.cobol.loader.Picture( "9999", False, 4, 0, 0, True, "V" )
        self.usage.setTypeInfo( self.picture )
        self.assertEqual( 2, self.usage.size() )
        self.assertEqual( "COMP", self.usage.source() )

class TestUsageComp3( unittest.TestCase ):
    def setUp( self ):
        self.usage = stingray.cobol.defs.UsageComp3( "COMP-3" )
        self.picture= stingray.cobol.loader.Picture( "9999999", False, 7, 0, 2, True, "V" )
    def test_should_show_size( self ):
        self.usage.setTypeInfo( self.picture )
        self.assertEqual( 4, self.usage.size() )
        self.assertEqual( "COMP-3", self.usage.source() )
```

Allocation

There are three kinds: group (i.e., a header under a group), successor, and redefines.

A Redefines object gets the offset information from another named element. A Group object for an item. A non-redefined element is “real”: it has a proper size and offset.

An item with a REDEFINES clause is an alias for another element. The offset comes from the other element. The size is reported as zero to simplify offset calculations.

Here’s a Mock DDE which can have a redefines clause, or be referenced by a redefines clause.

```
class MockDDE:
    def __init__( self, **kw ):
        self.children= []
        self.top= weakref.ref(self) # default
        self.__dict__.update( kw )
    def get( self, name ):
        return [c for c in self.children if c.name == name][0]
    def addChild( self, child ):
        self.children.append( child )
        child.parent= weakref.ref(self)
        child.top= self.top
```

There are two non-redefinies cases: Successor and Group. The DDE stands for itself. The size is as computed. The offset is as generated by the `cobol.defs.setSizeAndOffset()` function.

```
class TestAllocation_Group( unittest.TestCase ):
    def setUp( self ):
        self.parent= MockDDE( size=123, allocation=None,
            occurs=stingray.cobol.defs.Occurs(), totalSize=123 )
        self.group = stingray.cobol.defs.Group()
        self.dde= MockDDE( size=123, allocation=self.group,
            occurs=stingray.cobol.defs.Occurs(), totalSize=123 )
        self.parent.addChild( self.dde )
    def test_should_get_size_and_offset( self ):
        self.group.resolve( self.dde )
        self.assertEqual( 123, self.dde.allocation.totalSize() )
        self.assertEqual( 13, self.dde.allocation.offset(13) )
        self.assertIs( self.dde, self.dde.allocation.dde() )

class TestAllocation_Successor( unittest.TestCase ):
    def setUp( self ):
        self.parent= MockDDE( size=12, allocation=None,
            occurs=stingray.cobol.defs.Occurs(), totalSize=12 )
        self.prev= MockDDE( size=5,
            occurs=stingray.cobol.defs.Occurs(), totalSize=5 )
        self.parent.addChild( self.prev )
        self.successor = stingray.cobol.defs.Successor( self.prev )
        self.dde= MockDDE( size=7, allocation=self.successor, occurs=stingray.cobol.defs.Occurs(), to
        self.parent.addChild( self.dde )
    def test_should_get_size_and_offset( self ):
        self.successor.resolve( self.dde )
        self.assertEqual( 7, self.dde.allocation.totalSize() )
        self.assertEqual( 5, self.dde.allocation.offset(5) )
        self.assertIs( self.dde, self.dde.allocation.dde() )
```

Redefines is a reference to another DDE. The other DDE is located by the resolver pass. The size and offset come from the other element.

```
class TestAllocation_Rdefines( unittest.TestCase ):
    def setUp( self ):
        self.parent= MockDDE( name= "TOP", size=123, allocation=None, occurs=stingray.cobol.defs.Occu
        self.otherdde= MockDDE( name= "SOME-NAME", size=100, offset=23 )
        self.parent.addChild( self.otherdde )

        self.redefines_some_name = stingray.cobol.defs.Redefines( "SOME-NAME" )
        self.dde= MockDDE( name= "REDEF", size=47, allocation=self.redefines_some_name, occurs=sting
        self.parent.addChild( self.dde )
    def test_should_get_size_and_offset( self ):
        """Since this redefines something else, it doesn't contribute any size."""
        self.redefines_some_name.resolve( self.dde )
        self.assertEqual( 0, self.dde.allocation.totalSize() )
        self.assertEqual( 23, self.dde.allocation.offset(13) )
        self.assertIs( self.dde, self.dde.allocation.dde() )
```

Size And Offset Function

The idea is that this function does a depth-first traversal to accumulate the size in each group-level DDE.

Size and Offset Mocks

To test size and offset function, we need a number of mocks. We need to mock a DDE. Plus, we need to mock Usage and Redefines classes, also.

We also mock the iterable protocol of the DDE.

```
class MockDDE2:
    def __init__( self, **kw ):
        self.occurs= stingray.cobol.defs.Occurs()
        self.allocation= None
        self.children= []
        self.parent= None
        self.level= None
        self.name= "FILLER"
        self.picture= ""
        self.sizeScalePrecision= ()
        self.dimensionality= ()
        self.indent= 0
        self.__dict__.update( kw )
        self.size= self.usage.length
    def __repr__( self ):
        rc= ""
        if isinstance(self.allocation,stingray.cobol.defs.Redefines):
            rc= "REDEFINES {0}".format( self.allocation.name )
        oc= str(self.occurs)
        return "{0} {1} {2} {3} {4}.  {5} {6}".format(
            self.level, self.name, self.picture, rc, oc,
            self.offset, self.size,  )
    def addChild( self, child ):
        child.parent= weakref.ref(self)
        child.top= self.top
        self.children.append( child )
        child.indent= self.indent+1
    def get( self, name ):
        for d in self:
            if d.name == name: return d
    def pathTo( self ):
        return self.name
    def __iter__( self ):
        yield self
        for c in self.children:
            for c_i in c:
                yield c_i

class MockUsage:
    class MockCell:
        def __init__( self, buffer, workbook ):
            self.buffer= buffer
            self.workbook= workbook
    def __init__( self, source, **kw ):
        self.source_= source
        self.typeInfo= None
        self.length= 0
        self.__dict__.update( kw )
    def size( self ):
        return self.length
    def setTypeInfo( self, picture ):
        self.typeInfo= picture
        self.length= picture.length
    def source( self ):
        return self.source_
    def create_func( self, raw, workbook, attr ):
        return MockUsage.MockCell(raw, workbook, attr)


def mock_resolver( top ):
    for aDDE in top:
        if aDDE.allocation:
            aDDE.allocation.resolve( aDDE )
        else: # Not setup by test.
            #aDDE.allocation= stingray.cobol.defs.Group()
            pass
```

Size and Offset Cases

Exercise the size and offset function with a number of cases.

```
class TestFlatSizeOffset( unittest.TestCase ):
    def setUp( self ):
        self.top= MockDDE2( level='01', allocation=stingray.cobol.defs.Group(), picture=None, usage=M
        self.top.top= weakref.ref(self.top)
        self.top.addChild(
            MockDDE2( level='05', usage=MockUsage( "", length=3 ),
                allocation=stingray.cobol.defs.Group(), picture="XXX" ) )
        self.top.addChild(
            MockDDE2( level='05', usage=MockUsage( "", length=5 ),
                allocation=stingray.cobol.defs.Successor(self.top.children[-1]), picture="99999" ) )
        self.top.addChild(
            MockDDE2( level='05', usage=MockUsage( "", length=7 ),
                allocation=stingray.cobol.defs.Successor(self.top.children[-1]), picture="9999999" )
        mock_resolver( self.top )
    def test_should_set_size( self ):
        stingray.cobol.defs.setSizeAndOffset( self.top )
        self.assertEqual( 15, self.top.size )
        self.assertEqual( 0, self.top.children[0].offset )
        self.assertEqual( 3, self.top.children[0].size )
        self.assertEqual( 3, self.top.children[1].offset )
        self.assertEqual( 5, self.top.children[1].size )
        self.assertEqual( 8, self.top.children[2].offset )
        self.assertEqual( 7, self.top.children[2].size )

class TestNestedSizeOffset( unittest.TestCase ):
    def setUp( self ):
        self.top= MockDDE2( level='01', allocation=stingray.cobol.defs.Group(), picture=None, usage=M
        self.top.top= weakref.ref(self.top)
        self.top.parent= None
        self.top.addChild(
            MockDDE2( level='05', usage=MockUsage( "", length=3 ),
                allocation=stingray.cobol.defs.Group(), picture="XXX" ) )
        self.top.addChild(
            MockDDE2( level='05', usage=MockUsage( "", length=5 ),
                allocation=stingray.cobol.defs.Successor(self.top.children[-1]), picture="99999" ) )
        sub_group= MockDDE2( level='05',
            allocation=stingray.cobol.defs.Successor(self.top.children[-1]), usage=MockUsage("") )
        self.top.addChild( sub_group )
        sub_group.addChild(
            MockDDE2( level='10', usage=MockUsage( "", length=7 ),
                allocation=stingray.cobol.defs.Group(), picture="9999999" ) )
        sub_group.addChild(
            MockDDE2( level='10', usage=MockUsage( "", length=9 ),
                allocation=stingray.cobol.defs.Successor(self.top.children[-1]), picture="XXXXXXXXX"
        self.top.addChild(
            MockDDE2( level='05', usage=MockUsage( "", length=11 ),
                allocation=stingray.cobol.defs.Successor(self.top.children[-1]), picture="XXXXXXXXXXX
        mock_resolver( self.top )
    def test_should_set_size( self ):
        stingray.cobol.defs.setSizeAndOffset( self.top )
        self.assertEqual( 35, self.top.size )
        self.assertEqual( 0, self.top.children[0].offset )
        self.assertEqual( 3, self.top.children[0].size )
        self.assertEqual( 3, self.top.children[1].offset )
        self.assertEqual( 5, self.top.children[1].size )
        self.assertEqual( 8, self.top.children[2].offset )
        self.assertEqual( 16, self.top.children[2].size )
        self.assertEqual( 24, self.top.children[3].offset )
        self.assertEqual( 11, self.top.children[3].size )
        self.assertEqual( 8, self.top.children[2].children[0].offset )
        self.assertEqual( 7, self.top.children[2].children[0].size )
        self.assertEqual( 15, self.top.children[2].children[1].offset )
        self.assertEqual( 9, self.top.children[2].children[1].size )

class TestRedefinesSizeOffset( unittest.TestCase ):
    def setUp( self ):
        self.top= MockDDE2( level='01', allocation=stingray.cobol.defs.Group(), usage=MockUsage("") )
        self.top.top= weakref.ref(self.top)
        self.top.addChild(
            MockDDE2( level='05', usage=MockUsage( "", length=3 ),
                allocation=stingray.cobol.defs.Group(), picture="XXX" ) )
        sub_group_1= MockDDE2( level='05', name="GROUP-1",
            allocation=stingray.cobol.defs.Successor(self.top.children[-1]), usage=MockUsage("") )
        self.top.addChild( sub_group_1 )
        sub_group_2= MockDDE2( level='05', name="GROUP-2",
            allocation=stingray.cobol.defs.Redefines(refers_to=sub_group_1, name="GROUP-1",), usage=M
        self.top.addChild( sub_group_2 )
        self.top.addChild(
            MockDDE2( level='05', usage=MockUsage( "", length=5 ),
                allocation=stingray.cobol.defs.Successor(self.top.children[-1]), picture="XXXXX" ) )

        sub_group_1.addChild(
            MockDDE2( level='10', usage=MockUsage( "", length=5 ),
                allocation=stingray.cobol.defs.Group(), picture="99999" ) )
        sub_group_1.addChild(
            MockDDE2( level='10', usage=MockUsage( "", length=7 ),
                allocation=stingray.cobol.defs.Successor(sub_group_1.children[-1]), picture="9999999"
        sub_group_2.addChild(
            MockDDE2( level='10', usage=MockUsage( "", length=9 ),
                allocation=stingray.cobol.defs.Group(), picture="XXXXXXXXX" ) )
        sub_group_2.addChild(
            MockDDE2( level='10', usage=MockUsage( "", length=3 ),
                allocation=stingray.cobol.defs.Successor(sub_group_2.children[-1]), picture="XXX" ) )

        mock_resolver( self.top )
    def test_should_set_size( self ):
        stingray.cobol.defs.setSizeAndOffset( self.top )
        self.assertEqual( 20, self.top.size )
        self.assertEqual( 0, self.top.children[0].offset )
        self.assertEqual( 3, self.top.children[0].size )
        self.assertEqual( 3, self.top.children[1].offset )
        self.assertEqual( 12, self.top.children[1].size )
        self.assertEqual( 3, self.top.children[2].offset )
        self.assertEqual( 12, self.top.children[2].size )
        self.assertEqual( 15, self.top.children[3].offset )
        self.assertEqual( 5, self.top.children[3].size )
        self.assertEqual( 3, self.top.children[1].children[0].offset )
        self.assertEqual( 5, self.top.children[1].children[0].size )
        self.assertEqual( 8, self.top.children[1].children[1].offset )
        self.assertEqual( 7, self.top.children[1].children[1].size )
        self.assertEqual( 3, self.top.children[2].children[0].offset )
        self.assertEqual( 9, self.top.children[2].children[0].size )
        self.assertEqual( 12, self.top.children[2].children[1].offset )
        self.assertEqual( 3, self.top.children[2].children[1].size )

class TestOccursSizeOffset( unittest.TestCase ):
    def setUp( self ):
        self.top= MockDDE2( level='01', allocation=stingray.cobol.defs.Group(), picture=None, usage=M
        self.top.top= weakref.ref(self.top)
        self.top.addChild(
            MockDDE2( level='05', usage=MockUsage( "", length=3 ),
                allocation=stingray.cobol.defs.Group(), picture="XXX" ) )
        self.top.addChild(
            MockDDE2( level='05', usage=MockUsage( "", length=5 ),
                allocation=stingray.cobol.defs.Successor(self.top.children[-1]), picture="99999",
                occurs=stingray.cobol.defs.OccursFixed(7) ) )
        mock_resolver( self.top )
    def test_should_set_size( self ):
        stingray.cobol.defs.setSizeAndOffset( self.top )
        self.assertEqual( 38, self.top.totalSize )
        self.assertEqual( 0, self.top.children[0].offset )
        self.assertEqual( 3, self.top.children[0].size )
        self.assertEqual( 3, self.top.children[1].offset )
        self.assertEqual( 35, self.top.children[1].totalSize )
        self.assertEqual( 5, self.top.children[1].size )
        self.assertEqual( 7, self.top.children[1].occurs.number(None) )

class TestGroupOccursSizeOffset( unittest.TestCase ):
    def setUp( self ):
        self.top= MockDDE2( level='01', allocation=stingray.cobol.defs.Group(), picture=None, usage=M
        self.top.top= weakref.ref(self.top)
        self.top.addChild(
            MockDDE2( level='05', usage=MockUsage( "", length=3 ),
                allocation=stingray.cobol.defs.Group(), picture="XXX" ) )
        sub_group_1= MockDDE2( level='05', name="GROUP-1",
            allocation=stingray.cobol.defs.Successor(self.top.children[-1]), picture=None,
            occurs=stingray.cobol.defs.OccursFixed(4), usage=MockUsage("") )
        self.top.addChild( sub_group_1 )
        sub_group_1.addChild(
            MockDDE2( level=’10’, usage=MockUsage( "", length=5 ),
                allocation=stingray.cobol.defs.Group(), picture="99999",
                occurs=stingray.cobol.defs.OccursFixed(7) ) )
        mock_resolver( self.top )
    def test_should_set_size( self ):
        stingray.cobol.defs.setSizeAndOffset( self.top )
        self.assertEqual( 7*4*5+3, self.top.totalSize )
        self.assertEqual( 0, self.top.children[0].offset )
        self.assertEqual( 3, self.top.children[0].size )
        self.assertEqual( 3, self.top.children[1].offset )
        self.assertEqual( 7*4*5, self.top.children[1].totalSize )
        self.assertEqual( 7*5, self.top.children[1].size )
        self.assertEqual( 4, self.top.children[1].occurs.number(None) )
        self.assertEqual( 3, self.top.children[1].children[0].offset )
        self.assertEqual( 35, self.top.children[1].children[0].totalSize )
        self.assertEqual( 5, self.top.children[1].children[0].size )
        self.assertEqual( 7, self.top.children[1].children[0].occurs.number(None) )
    def test_should_be_repeatable( self ):
        stingray.cobol.defs.setSizeAndOffset( self.top )
        self.assertEqual( 7*4*5+3, self.top.totalSize )
        stingray.cobol.defs.setSizeAndOffset( self.top )
        self.assertEqual( 7*4*5+3, self.top.totalSize )
        stingray.cobol.defs.setSizeAndOffset( self.top )
        self.assertEqual( 7*4*5+3, self.top.totalSize )
```

Resolve Redefines

We need to test the `cobol.loader.resolver()` function.

```
class TestRedefinesNameResolver( unittest.TestCase ):
    def setUp( self ):
        self.top= MockDDE2( level=’01’, allocation=stingray.cobol.defs.Group(), picture=None, usage=M
        self.top.top= weakref.ref(self.top)
        self.top.addChild(
            MockDDE2( level=’05’, usage=MockUsage( "", length=3 ),
                allocation=stingray.cobol.defs.Group(), picture="XXX" ) )
        self.sub_group_1= MockDDE2( level=’05’, name="GROUP-1",
            allocation=stingray.cobol.defs.Successor(self.top.children[-1]), usage=MockUsage("") )
        self.top.addChild( self.sub_group_1 )
        self.sub_group_2= MockDDE2( level=’05’, name="GROUP-2",
            allocation=stingray.cobol.defs.Redefines(refers_to=None, name="GROUP-1",), usage=MockUsag
        self.top.addChild( self.sub_group_2 )
        self.top.addChild(
            MockDDE2( level=’05’, usage=MockUsage( "", length=5 ),
                allocation=stingray.cobol.defs.Successor(self.top.children[-1]), picture="XXXXX" ) )

        self.sub_group_1.addChild(
            MockDDE2( level=’10’, usage=MockUsage( "",length=5 ),
                allocation=stingray.cobol.defs.Group(), picture="99999" ) )
        self.sub_group_1.addChild(
            MockDDE2( level=’10’, usage=MockUsage( "", length=7 ),
                allocation=stingray.cobol.defs.Successor(self.sub_group_1.children[-1]), picture="99
        self.sub_group_2.addChild(
            MockDDE2( level='10', usage=MockUsage( "", length=9 ),
                allocation=stingray.cobol.defs.Group(), picture="XXXXXXXXX" ) )
        self.sub_group_2.addChild(
            MockDDE2( level='10', usage=MockUsage( "", length=3 ),
                allocation=stingray.cobol.defs.Successor(self.sub_group_2.children[-1]), picture="XXX

        self.top.search= {
            'GROUP-1': self.sub_group_1,
            'GROUP-2': self.sub_group_2,
            }

    def test_should_resolve( self ):
        stingray.cobol.defs.resolver( self.top )
        self.assertIs( self.sub_group_1, self.sub_group_2.allocation.refers_to )
```

Set Dimensionality

The `stingray.cobol.defs.setDimensionality()` function pushes the OCCURS information down to each child of the OCCURS. This builds the effective dimensionality of the lowest-level elements.

```
* COPY3.COB
 01  SURVEY-RESPONSES.
     05  QUESTION-NUMBER         OCCURS 10 TIMES.
         10  RESPONSE-CATEGORY     OCCURS 3 TIMES.
             15  ANSWER                          PIC 99.

class Test_Dimensionality( unittest.TestCase ):
    def setUp( self ):
        self.top= MockDDE2( level='01', name='SURVEY-RESPONSES',
            allocation=stingray.cobol.defs.Group(), picture=None, offset=0, size=60,
            occurs=stingray.cobol.defs.Occurs(), usage=MockUsage("") )
        self.top.top= weakref.ref(self.top)
        self.top.parent= None
        self.group_05 = MockDDE2( level='05', name='QUESTION-NUMBER',
            allocation=stingray.cobol.defs.Group(), usage=MockUsage(""),
            occurs=stingray.cobol.defs.OccursFixed(10), offset=0, totalSize=60, size=6 )
        self.top.addChild( self.group_05 )
        self.group_10 = MockDDE2( level='10', name='RESPONSE-CATEGORY',
            allocation=stingray.cobol.defs.Group(), usage=MockUsage(""),
            occurs=stingray.cobol.defs.OccursFixed(3), offset=0, totalSize=6, size=2 )
        self.group_05.addChild( self.group_10 )
        self.group_15 = MockDDE2( level='15', name='ANSWER',
            allocation=stingray.cobol.defs.Group(), picture="99",
            occurs=stingray.cobol.defs.Occurs(), offset=0, totalSize=2, size=2, usage=MockUsage("99")
        self.group_10.addChild( self.group_15 )
    def test_should_set_dimensions( self ):
        stingray.cobol.defs.setDimensionality( self.top )
        self.assertEqual( 1, self.top.occurs.number(None) )
        self.assertEqual( (), self.top.dimensionality )
        self.assertEqual( 10, self.group_05.occurs.number(None) )
        self.assertEqual( (self.group_05,), self.group_05.dimensionality )
        self.assertEqual( 3, self.group_10.occurs.number(None) )
        self.assertEqual( (self.group_05,self.group_10), self.group_10.dimensionality )
        self.assertEqual( 1, self.group_15.occurs.number(None) )
        self.assertEqual( (self.group_05,self.group_10), self.group_15.dimensionality )
```

DDE Construction Methods

DDE class has many methods. They fit into three functionality categories.

1. Building the DDE, adding children, visiting.
2. Getting elements by simple name. Computing path names. Getting elements by path name.
3. Getting a run-time value from a buffer, accounting for indexes. This is DDE Access and is tested separately.

The first two areas are tested here.

```
class TestDDEMethods( unittest.TestCase ):
    def setUp( self ):
        self.top= stingray.cobol.defs.DDE( level=’01’, name=’TOP’, usage=MockUsage("") )
        self.top.top= weakref.ref(self.top)
        self.group= stingray.cobol.defs.DDE( level=’05’, name=’GROUP’, usage=MockUsage("")  )
        self.element= stingray.cobol.defs.DDE( level=’10’, name=’ELEMENT’, pic=’X(5)’, usage=MockUsag
        self.top.addChild( self.group )
        self.group.addChild( self.element )
    def test_structure( self ):
        self.assertIs( self.group, self.top.children[0] )
        self.assertIs( self.element, self.group.children[0] )
    def test_path_name( self ):
        self.assertEqual( "TOP.GROUP.ELEMENT", self.element.pathTo() )
        self.assertIs( self.element, self.top.getPath("TOP.GROUP.ELEMENT") )
    def test_get( self ):
        g= self.top.get( "GROUP" )
        self.assertIs( self.group, g )
        e= g.get( "ELEMENT" )
        self.assertIs( self.element, e )
```

Parsing Single DDE

The parser handles 11 different clauses. Additionally, it makes a single DDE element as well as making a composite DDE record. A parser depends on a lexical scanner. However, since our scanner is essentially an iterator, we don’t need to do very much to mock it.

To test just the `cobol.loader.RecordFactory` in isolation, we need to provide mocks for a large number of dependenices.

```
self.redefines_class= Redefines
self.non_redefines_class= NonRedefines
self.display_class= UsageDisplay
self.comp_class= UsageComp
self.comp3_class= UsageComp3
```

Note that there are numerous syntax errors which we do not test for. Ideally, a DDE clause is used in working software, and passes through a COBOL compiler. This isn’t *lint for COBOL*.

We have three test cases: things the parser finds (and keeps), things the parser skips gracefully, and things we can’t cope with.

```
class TestParser( unittest.TestCase ):
    def setUp( self ):
        self.parser= stingray.cobol.loader.RecordFactory()
        #self.parser.redefines_class= MockRedefines
        #self.parser.non_redefines_class= MockNonRedefine
        self.parser.display_class= MockUsage
        self.parser.comp_class= MockUsage
        self.parser.comp3_class= MockUsage
    def test_should_parse_group( self ):
        source= ( "01", "GROUP", "." )
        dde= next(self.parser.dde_iter(iter(source)))
        self.assertEqual( ’01’, dde.level )
        self.assertEqual( ’GROUP’, dde.name )
        self.assertEqual( 0, len(dde.children) )
        self.assertEqual( 1, dde.occurs.number(None) )
        self.assertIsNone( dde.picture )
        self.assertIsNone( dde.allocation )
        self.assertEqual( "", dde.usage.source() )
    def test_should_parse_filler( self ):
        source= ( "05", "PIC", "X(10)", "." )
        dde= next(self.parser.dde_iter(iter(source)))
        self.assertEqual( ’05’, dde.level )
        self.assertEqual( ’FILLER’, dde.name )
        self.assertEqual( 0, len(dde.children) )
        self.assertEqual( 1, dde.occurs.number(None) )
        self.assertEqual( "X(10)", dde.picture )
        self.assertIsNone( dde.allocation )
        self.assertEqual( "", dde.usage.source() )
    def test_should_parse_name( self ):
        source= ( "05", "ELEMENTARY", "PIC", "X(10)", "." )
        dde= next(self.parser.dde_iter(iter(source)))
        self.assertEqual( ’05’, dde.level )
        self.assertEqual( ’ELEMENTARY’, dde.name )
        self.assertEqual( 0, len(dde.children) )
        self.assertEqual( 1, dde.occurs.number(None) )
        self.assertEqual( "X(10)", dde.picture )
        self.assertIsNone( dde.allocation )
        self.assertEqual( "", dde.usage.source() )
    def test_should_parse_occurs( self ):
        src1= ( "05", "ELEMENTARY-OCCURS", "PIC", "S9999", "OCCURS", "5", "TIMES", "." )
        dde= next(self.parser.dde_iter(iter(src1)))
        self.assertEqual( ’05’, dde.level )
        self.assertEqual( ’ELEMENTARY-OCCURS’, dde.name )
        self.assertEqual( 0, len(dde.children) )
        self.assertEqual( 5, dde.occurs.number(None) )
        self.assertEqual( "S9999", dde.picture )
        self.assertIsNone( dde.allocation )
        self.assertEqual( "", dde.usage.source() )
        src2= ( "05", "GROUP-OCCURS", "OCCURS", "7", "TIMES", "INDEXED", "BY", "IRRELEVANT", "." )
        dde= next(self.parser.dde_iter(iter(src2)))
        self.assertEqual( ’05’, dde.level )
        self.assertEqual( ’GROUP-OCCURS’, dde.name )
        self.assertEqual( 0, len(dde.children) )
        self.assertEqual( 7, dde.occurs.number(None) )
        self.assertIsNone( dde.picture )
        self.assertIsNone( dde.allocation )
        self.assertEqual( "", dde.usage.source() )
    def test_should_parse_picture( self ):
        """Details of picture clause tested separately."""
        src1= ( "05", "ELEMENTARY-PIC", "PIC", "S9999.999", "." )
        dde= next(self.parser.dde_iter(iter(src1)))
        self.assertEqual( ’05’, dde.level )
        self.assertEqual( ’ELEMENTARY-PIC’, dde.name )
        self.assertEqual( 0, len(dde.children) )
        self.assertEqual( 1, dde.occurs.number(None) )
        self.assertEqual( "S9999.999", dde.picture )
        self.assertIsNone( dde.allocation )
        self.assertEqual( "", dde.usage.source() )
    def test_should_parse_redefines( self ):
        src1= ( "05", "ELEMENTARY-REDEF", "PIC", "S9999", "REDEFINES", "SOME-NAME", "." )
        dde= next(self.parser.dde_iter(iter(src1)))
        self.assertEqual( ’05’, dde.level )
        self.assertEqual( ’ELEMENTARY-REDEF’, dde.name )
        self.assertEqual( 0, len(dde.children) )
        self.assertEqual( 1, dde.occurs.number(None) )
        self.assertEqual( "S9999", dde.picture )
        self.assertEqual( "SOME-NAME", dde.allocation.name )
        self.assertEqual( "", dde.usage.source() )
        src2= ( "05", "GROUP-REDEF", "OCCURS", "7", "TIMES", "REDEFINES", "ANOTHER-NAME", "." )
        dde= next(self.parser.dde_iter(iter(src2)))
        self.assertEqual( ’05’, dde.level )
        self.assertEqual( ’GROUP-REDEF’, dde.name )
        self.assertEqual( 0, len(dde.children) )
        self.assertEqual( 7, dde.occurs.number(None) )
        self.assertIsNone( dde.picture )
        self.assertEqual( "ANOTHER-NAME", dde.allocation.name )
        self.assertEqual( "", dde.usage.source() )
    def test_should_parse_usage( self ):
        src1= ( "05", "USE-DISPLAY", "PIC", "S9999", "USAGE", "DISPLAY", "." )
        dde= next(self.parser.dde_iter(iter(src1)))
        self.assertEqual( ’05’, dde.level )
        self.assertEqual( ’USE-DISPLAY’, dde.name )
        self.assertEqual( 0, len(dde.children) )
        self.assertEqual( 1, dde.occurs.number(None) )
        self.assertEqual( "S9999", dde.picture )
        self.assertIsNone( dde.allocation )
        self.assertEqual( "DISPLAY", dde.usage.source() )
        src2= ( "05", "USE-COMP-3", "PIC", "S9(7)V99", "USAGE", "COMP-3", "." )
        dde= next(self.parser.dde_iter(iter(src2)))
        self.assertEqual( ’05’, dde.level )
        self.assertEqual( ’USE-COMP-3’, dde.name )
        self.assertEqual( 0, len(dde.children) )
        self.assertEqual( 1, dde.occurs.number(None) )
        self.assertEqual( "S9(7)V99", dde.picture )
        self.assertIsNone( dde.allocation )
        self.assertEqual( "COMP-3", dde.usage.source() )
        src3= ( "05", "USE-COMP-3-ALT", "PIC", "S9(9)V99", "COMP-3", "." )
        dde= next(self.parser.dde_iter(iter(src3)))
        self.assertEqual( ’05’, dde.level )
        self.assertEqual( ’USE-COMP-3-ALT’, dde.name )
        self.assertEqual( 0, len(dde.children) )
        self.assertEqual( 1, dde.occurs.number(None) )
        self.assertEqual( "S9(9)V99", dde.picture )
        self.assertIsNone( dde.allocation )
        self.assertEqual( "COMP-3", dde.usage.source() )
    def test_should_parse_depending_on_1( self ):
        src1= ( "05", "DEP-1", "OCCURS", "1", "TO", "5", "TIMES", "DEPENDING", "ON", "ODO-1", "." )
        self.parser.lex= iter(src1)
        dde= next(self.parser.dde_iter(iter(src1)))
        self.assertEqual( ’05’, dde.level )
        self.assertEqual( ’DEP-1’, dde.name )
        self.assertEqual( 0, len(dde.children) )
        self.assertEqual( "ODO-1", dde.occurs.name ) # number(None) )
        self.assertEqual( None, dde.picture )
        self.assertIsNone( dde.allocation )
    def test_should_parse_depending_on_2( self ):
        src1= ( "05", "DEP-2", "OCCURS", "TO", "5", "TIMES", "DEPENDING", "ON", "ODO-2", "." )
        self.parser.lex= iter(src1)
        dde= next(self.parser.dde_iter(iter(src1)))
        self.assertEqual( ’05’, dde.level )
        self.assertEqual( ’DEP-2’, dde.name )
        self.assertEqual( 0, len(dde.children) )
        self.assertEqual( "ODO-2", dde.occurs.name ) # number(None) )
        self.assertEqual( None, dde.picture )
        self.assertIsNone( dde.allocation )
```

Syntax which is silently skipped.

```
class TestParserSkip( unittest.TestCase ):
    def setUp( self ):
        self.parser= stingray.cobol.loader.RecordFactory()
    def test_should_skip_blank( self ):
        src1= ( "05", "BLANK-ZERO-1", "PIC", "X(10)", "BLANK", "ZERO", "." )
        dde= next(self.parser.dde_iter(iter(src1)))
        self.assertEqual( ’05’, dde.level )
        self.assertEqual( ’BLANK-ZERO-1’, dde.name )
        self.assertEqual( 0, len(dde.children) )
        self.assertEqual( 1, dde.occurs.number(None) )
        self.assertEqual( "X(10)", dde.picture )
        self.assertIsNone( dde.allocation )
        self.assertEqual( "", dde.usage.source() )
        self.assertIsNone( dde.initValue )
        src2= ( "05", "BLANK-ZERO-2", "PIC", "X(10)", "BLANK", "WHEN", "ZEROES", "." )
        dde= next(self.parser.dde_iter(iter(src2)))
        self.assertEqual( ’05’, dde.level )
        self.assertEqual( ’BLANK-ZERO-2’, dde.name )
        self.assertEqual( 0, len(dde.children) )
        self.assertEqual( 1, dde.occurs.number(None) )
        self.assertEqual( "X(10)", dde.picture )
        self.assertIsNone( dde.allocation )
        self.assertEqual( "", dde.usage.source() )
        self.assertIsNone( dde.initValue )
    def test_should_skip_justified( self ):
        src1= ( "05", "JUST-RIGHT-1", "PIC", "X(10)", "JUST", "RIGHT", "." )
        dde= next(self.parser.dde_iter(iter(src1)))
        self.assertEqual( ’05’, dde.level )
        self.assertEqual( ’JUST-RIGHT-1’, dde.name )
        self.assertEqual( 0, len(dde.children) )
        self.assertEqual( 1, dde.occurs.number(None) )
        self.assertEqual( "X(10)", dde.picture )
        self.assertIsNone( dde.allocation )
        self.assertEqual( "", dde.usage.source() )
        self.assertIsNone( dde.initValue )
        src2= ( "05", "JUST-RIGHT-2", "PIC", "X(10)", "JUSTIFIED", "RIGHT", "." )
        dde= next(self.parser.dde_iter(iter(src2)))
        self.assertEqual( ’05’, dde.level )
        self.assertEqual( ’JUST-RIGHT-2’, dde.name )
        self.assertEqual( 0, len(dde.children) )
        self.assertEqual( 1, dde.occurs.number(None) )
        self.assertEqual( "X(10)", dde.picture )
        self.assertIsNone( dde.allocation )
        self.assertEqual( "", dde.usage.source() )
        self.assertIsNone( dde.initValue )
    def test_should_skip_value( self ):
        src1= ( "05", "VALUE-1", "PIC", "X(8)", "VALUE", "'10 CHARS'", "." )
        dde= next(self.parser.dde_iter(iter(src1)))
        self.assertEqual( '05', dde.level )
        self.assertEqual( 'VALUE-1', dde.name )
        self.assertEqual( 0, len(dde.children) )
        self.assertEqual( 1, dde.occurs.number(None) )
        self.assertEqual( "X(8)", dde.picture )
        self.assertIsNone( dde.allocation )
        self.assertEqual( "", dde.usage.source() )
        self.assertIsNone( dde.initValue )
```

Todo

Test EXTERNAL, GLOBAL as Skipped Words, too.

A few clauses may be relevant for some kinds of DDE's. These can impact the encoding of the bytes. We don't parse them, however, because they are rarely seen in the wild.

```
class TestParserException( unittest.TestCase ):
    def setUp( self ):
        self.parser= stingray.cobol.loader.RecordFactory()
    def test_should_fail_renames( self ):
        src1= ( "05", "BLANK-ZERO-1", "PIC", "X(10)", "RENAMES", "SOME-NAME", "." )
        try:
            dde= next(self.parser.dde_iter(iter(src1)))
            self.fail( "Should not parse" )
        except stingray.cobol.defs.UnsupportedError as e:
            pass
    def test_should_fail_sign( self ):
        src1= ( "05", "SIGN-1", "PIC", "X(10)", "LEADING", "SIGN", "." )
        self.parser.lex= iter(src1)
        try:
            dde= next(self.parser.dde_iter(iter(src1)))
            self.fail( "Should not parse" )
        except stingray.cobol.defs.UnsupportedError as e:
            pass
        src2= ( "05", "SIGN-2", "PIC", "X(10)", "TRAILING", "." )
        self.parser.lex= iter(src2)
        try:
            dde= next(self.parser.dde_iter(iter(src1)))
            self.fail( "Should not parse" )
        except stingray.cobol.defs.UnsupportedError as e:
            pass
        src3= ( "05", "SIGN-3", "PIC", "X(10)", "SIGN", "IS", "SEPARATE", "." )
        self.parser.lex= iter(src3)
        try:
            dde= next(self.parser.dde_iter(iter(src1)))
            self.fail( "Should not parse" )
        except stingray.cobol.defs.UnsupportedError as e:
            pass
    def test_should_fail_synchronized( self ):
        src1= ( "05", "SYNC-1", "PIC", "X(10)", "SYNCHRONIZED", "." )
        self.parser.lex= iter(src1)
        try:
            dde= next(self.parser.dde_iter(iter(src1)))
            self.fail( "Should not parse" )
        except stingray.cobol.defs.UnsupportedError as e:
            pass
```

Parsing Complete DDE

Parsing a complete DDE is a multi-step dance. First, parse all the clauses. Then apply some standard functions to resolve the REDEFINES clauses as well as compute sizes and offsets.

To test just the `cobol.loader.RecordFactory` in isolation, we need to provide mocks for a large number of dependenices.

```
# Built during the parsing
self.redefines_class= Redefines
self.non_redefines_class= NonRedefines
self.display_class= UsageDisplay
self.comp_class= UsageComp
self.comp3_class= UsageComp3
```

We use dependency injection here to tease apart Redefines and Usage classes. We can use existing mocks for this.

Here's a test of the "main" method for parsing a record. Note that `stingray.cobol.loader.RecordFactory.makeRecord()` is iterable, so we make a list and take the first element.

We could as easily `next()` it to get the first element.

Test the parser

```
class TestCompleteParser( unittest.TestCase ):
    def setUp( self ):
        self.parser= stingray.cobol.loader.RecordFactory()
        self.parser.display_class= MockUsage
        self.parser.comp_class= MockUsage
        self.parser.comp3_class= MockUsage
    def test_should_parse( self ):
        copy1= """
              * COPY1.COB
               01  DETAIL-LINE.
                   05                              PIC X(7).
                   05  QUESTION                    PIC ZZ.
                   05                              PIC X(6).
                   05  PRINT-YES                   PIC ZZ.
"""
        self.lex= ['01', 'DETAIL-LINE', '.',
        '05', 'PIC', 'X(7)', '.',
        '05', 'QUESTION', 'PIC', 'ZZ', '.',
        '05', 'PIC', 'X(6)', '.',
        '05', 'PRINT-YES', 'PIC', 'ZZ', '.']
        dde= list(self.parser.makeRecord( iter( self.lex ) ))[0]
        self.assertEqual( "01", dde.top().level )
        self.assertEqual( "DETAIL-LINE", dde.top().name )
        self.assertEqual( 4, len(dde.top().children) )
        self.assertEqual( "", dde.top().usage.source() )
        c0= dde.top().children[0]
        self.assertEqual( "05", c0.level )
        self.assertEqual( "FILLER", c0.name )
        self.assertEqual( "X(7)", c0.picture )
        self.assertEqual( "", c0.usage.source() )
        self.assertEqual( 1, c0.occurs.number(None) )
        c1= dde.top().children[1]
        self.assertEqual( "05", c1.level )
        self.assertEqual( "QUESTION", c1.name )
        self.assertEqual( "ZZ", c1.picture )
        self.assertEqual( "", c1.usage.source() )
        self.assertEqual( 1, c1.occurs.number(None) )
        c2= dde.top().children[2]
        self.assertEqual( "05", c2.level )
        self.assertEqual( "FILLER", c2.name )
        self.assertEqual( "X(6)", c2.picture )
        self.assertEqual( "", c2.usage.source() )
        self.assertEqual( 1, c2.occurs.number(None) )
        c3= dde.top().children[3]
        self.assertEqual( "05", c3.level )
        self.assertEqual( "PRINT-YES", c3.name )
        self.assertEqual( "ZZ", c3.picture )
        self.assertEqual( "", c3.usage.source() )
        self.assertEqual( 1, c3.occurs.number(None) )
        self.assertEqual( 5, len(list(dde) ) )
```

Schema Maker

A DDE is tranformed into a flat schema from the highly-nested DDE structure via the `stingray.cobol.loader.make_schema()` function.

```
class TestNestedSchemaMaker( unittest.TestCase ):
    def setUp( self ):
        self.top= MockDDE2( level='01', name="TOP",
            allocation=stingray.cobol.defs.Group(), picture=None, size=35, offset=0, usage=MockUsage
        self.top.top= weakref.ref(self.top)
        self.top.addChild(
            MockDDE2( level='05', name="FIRST", usage=MockUsage( "", length=3 ),
                allocation=stingray.cobol.defs.Group(), picture="XXX", size=3, offset=0 ) )
        self.top.addChild(
            MockDDE2( level='05', name="SECOND", usage=MockUsage( "", length=5 ),
                allocation=stingray.cobol.defs.Successor(self.top.children[-1]), picture="99999", si
        sub_group= MockDDE2( level='05', name="GROUP",
            allocation=stingray.cobol.defs.Group(), picture=None, size=16, offset=8, usage=MockUsage
        self.top.addChild( sub_group )
        sub_group.addChild(
            MockDDE2( level='10', name="INNER", usage=MockUsage( "", length=7 ),
                allocation=stingray.cobol.defs.Group(), picture="9999999", size=8, offset=7 ) )
        sub_group.addChild(
            MockDDE2( level='10', usage=MockUsage( "", length=9 ),
                allocation=stingray.cobol.defs.Successor(sub_group.children[-1]), picture="XXXXXXXXX"
        self.top.addChild(
            MockDDE2( level='05', name="LAST", usage=MockUsage( "", length=11 ),
                allocation=stingray.cobol.defs.Successor(sub_group.children[-1]), picture="XXXXXXXXXX
    def test_should_set_size( self ):
        stingray.cobol.defs.setSizeAndOffset( self.top )
        schema = stingray.cobol.loader.make_schema( [self.top] )
        #DEBUG# print( schema )
        self.assertEqual( "TOP", schema[0].name )
        self.assertEqual( 0, schema[0].position )
        self.assertEqual( 0, schema[0].offset )
        self.assertEqual( 35, schema[0].size )
        self.assertEqual( "FIRST", schema[1].name )
        self.assertEqual( 1, schema[1].position )
        self.assertEqual( 0, schema[1].offset )
        self.assertEqual( 3, schema[1].size )
        self.assertEqual( "SECOND", schema[2].name )
        self.assertEqual( 2, schema[2].position )
        self.assertEqual( 3, schema[2].offset )
        self.assertEqual( 5, schema[2].size )
        self.assertEqual( "LAST", schema[6].name )
        self.assertEqual( 6, schema[6].position )
        self.assertEqual( 24, schema[6].offset )
        self.assertEqual( 11, schema[6].size )
```

Test Suite and Runner

In case we want to build up a larger test suite, we avoid doing any real work unless this is the main module being executed.

```
import test
suite= test.suite_maker( globals() )

if __name__ == "__main__":
    print( __file__ )
    unittest.TextTestRunner(verbosity=1).run(suite())
```

1.14.9 COBOL Integration Tests

These tests use the unittest framework, but don't test small fixtures in isolation. These are "integration" tests of the `cobol` and `cobol.loader` modules.

See *The COBOL Package* and *COBOL Loader Module – Parse COBOL Source to Load a Schema*.

Overheads

```
"""stingray.cobol Integration Tests.
This tests the cobol and cobol loader without mocked objects.
"""
import unittest
import decimal
import stingray.cobol.loader
import logging, sys
```

Superclass For Tests

This is a handy superclass for all the various tests. It refactors the `setUp()` method to assure that all of the tests have a common fixture.

```
class DDE_Test( unittest.TestCase ):
    def setUp( self ):
        self.lexer= stingray.cobol.loader.Lexer()
        self.rf= stingray.cobol.loader.RecordFactory()
```

DDE Test copybook 1 with basic features

Some basic COBOL.

```
copy1= """
      * COPY1.COB
       01  DETAIL-LINE.
           05                              PIC X(7).
           05  QUESTION                    PIC ZZ.
           05                              PIC X(6).
           05  PRINT-YES                   PIC ZZ.
           05                              PIC X(3).
           05  PRINT-NO                    PIC ZZ.
           05                              PIC X(6).
           05  NOT-SURE                    PIC ZZ.
           05                              PIC X(7).
"""
```

Be sure it parses. Be sure we can extract data.

```
class Test_Copybook_1( DDE_Test ):
    def setUp( self ):
        super().setUp()
        self.dde1 = list(self.rf.makeRecord( self.lexer.scan(copy1) ))[0]
        #stingray.cobol.loader.report( self.dde1 )
    def test_should_parse( self ):
        dde1= self.dde1
        self.assertEqual( 7, dde1.get( "QUESTION" ).offset )
        self.assertEqual( 2, dde1.get( "QUESTION" ).size )
        self.assertEqual( "ZZ", dde1.get( "QUESTION" ).sizeScalePrecision.final  )
        self.assertEqual( "", dde1.get( "QUESTION" ).usage.source() )
        self.assertEqual( 15, dde1.get( "PRINT-YES" ).offset )
        self.assertEqual( 2, dde1.get( "PRINT-YES" ).size )
        self.assertEqual( "ZZ", dde1.get( "PRINT-YES" ).sizeScalePrecision.final  )
        self.assertEqual( 20, dde1.get( "PRINT-NO" ).offset )
        self.assertEqual( 2, dde1.get( "PRINT-NO" ).size )
        self.assertEqual( "ZZ", dde1.get( "PRINT-NO" ).sizeScalePrecision.final  )
        self.assertEqual( 28, dde1.get( "NOT-SURE" ).offset )
        self.assertEqual( 2, dde1.get( "NOT-SURE" ).size )
        self.assertEqual( "ZZ", dde1.get( "NOT-SURE" ).sizeScalePrecision.final  )
    def test_should_extract( self ):
        schema = stingray.cobol.loader.make_schema( [self.dde1] )
        #print( schema )
        schema_dict= dict( (a.name, a) for a in schema )
        data= stingray.cobol.Character_File( name="",
            file_object= ["ABCDEFG01HIJKLM02OPQ03RSTUVW04YZabcde",],
            schema=schema )

        row= next( data.sheet( "" ).rows() )
        #stingray.cobol.dump( schema, row )

        self.assertEqual( "1", row.cell(schema_dict['QUESTION']).to_str() )
        self.assertEqual( 2, row.cell(schema_dict['PRINT-YES']).to_int() )
        self.assertEqual( 3, row.cell(schema_dict['PRINT-NO']).to_float() )
        self.assertEqual( decimal.Decimal('4'), row.cell(schema_dict['NOT-SURE']).to_decimal() )
```

DDE Test copybook 2 with 88-level item

Include 88-level items in the source.

```
copy2= """
      * COPY2.COB
       01  WORK-AREAS.
           05  ARE-THERE-MORE-RECORDS      PIC X(3)    VALUE ’YES’.
               88  NO-MORE-RECORDS                     VALUE ’NO ’.
           05  ANSWER-SUB                  PIC 99.
           05  QUESTION-SUB                PIC 99.
"""
```

Be sure it parses. Be sure we can extract data.

```
class Test_Copybook_2( DDE_Test ):
    def setUp( self ):
        super().setUp()
        self.dde2= list(self.rf.makeRecord( self.lexer.scan(copy2) ))[0]
        #stingray.cobol.loader.report( self.dde2 )
    def test_should_parse( self ):
        dde2= self.dde2
        self.assertEqual( 0, dde2.get("ARE-THERE-MORE-RECORDS").offset )
        self.assertEqual( 3, dde2.get("ARE-THERE-MORE-RECORDS").size )
        self.assertEqual( "XXX", dde2.get("ARE-THERE-MORE-RECORDS").sizeScalePrecision.final  )
        self.assertEqual( 0, dde2.get("NO-MORE-RECORDS").offset )
        self.assertEqual( 3, dde2.get("NO-MORE-RECORDS").size )
        self.assertEqual( 3, dde2.get("ANSWER-SUB").offset )
        self.assertEqual( 5, dde2.get("QUESTION-SUB").offset )
    def test_should_extract( self ):
        schema = stingray.cobol.loader.make_schema( [self.dde2] )
        schema_dict= dict( (a.name, a) for a in schema )
        data= stingray.cobol.Character_File( name="",
            file_object= ["NO 4567",],
            schema=schema, )

        row= next( data.sheet( "" ).rows() )
        #stingray.cobol.dump( schema, row )
        self.assertEqual( "NO", row.cell(schema_dict["ARE-THERE-MORE-RECORDS"]).to_str() )
        self.assertEqual( "NO", row.cell(schema_dict["NO-MORE-RECORDS"]).to_str() )
```

DDE Test copybook 3 with nested occurs level

This is a common two-dimensional COBOL structure.

```
copy3= """
      * COPY3.COB
       01  SURVEY-RESPONSES.
           05  QUESTION-NUMBER         OCCURS 10 TIMES.
               10  RESPONSE-CATEGORY     OCCURS 3 TIMES.
                   15  ANSWER                          PIC 99.
"""
```

Be sure that the various access methods (via Attribute and via Python tuple-of-tuples) all work.

```
class Test_Copybook_3( DDE_Test ):
    def setUp( self ):
        super().setUp()
        self.dde3= list(self.rf.makeRecord( self.lexer.scan(copy3) ))[0]
        #stingray.cobol.loader.report( self.dde3 )
    def test_should_extract( self ):
        schema = stingray.cobol.loader.make_schema( [self.dde3] )
        schema_dict= dict( (a.name, a) for a in schema )
        data = stingray.cobol.Character_File( name="",
            file_object= ["111213212223313233414243515253616263717273818283919293010203",],
            schema=schema )

        row= next( data.sheet( "" ).rows() )
        #stingray.cobol.dump( schema, row )
        self.assertEqual( 12, row.cell(schema_dict.get(’ANSWER’).index(1-1,2-1)).to_int() )
        self.assertEqual( 21, row.cell( schema_dict.get(’ANSWER’).index(2-1,1-1)).to_int() )
        self.assertEqual( 21, row.cell( schema_dict.get(’ANSWER’).index(1-1,4-1)).to_int() )
        try:
            self.assertEqual( 21, row.cell( schema_dict.get(’ANSWER’).index(1))[4].to_int() )
            self.fail()
        except IndexError as e:
            pass
```

DDE Test copybook 4 from page 174 with nested occurs level

From IBM COBOL Language Reference Manual, fourth edition: SC26-9046-03.

```
page174= """
       01 TABLE-RECORD.
          05 EMPLOYEE-TABLE OCCURS 10 TIMES
                ASCENDING KEY IS WAGE-RATE EMPLOYEE-NO
                INDEXED BY A, B.
             10 EMPLOYEE-NAME PIC X(20).
             10 EMPLOYEE-NO PIC 9(6).
             10 WAGE-RATE PIC 9999V99.
             10 WEEK-RECORD OCCURS 52 TIMES
                   ASCENDING KEY IS WEEK-NO INDEXED BY C.
                15 WEEK-NO PIC 99.
                15 AUTHORIZED-ABSENCES PIC 9.
                15 UNAUTHORIZED-ABSENCES PIC 9.
                15 LATE-ARRIVALS PIC 9.
"""
```

Be sure it parses. There’s nothing novel in the structure, but the syntax has numerous things we need to gracefully skip.

```
class Test_Copybook_4( DDE_Test ):
    def setUp( self ):
        super().setUp()
        self.dde4= list(self.rf.makeRecord( self.lexer.scan(page174) ))[0]
        #stingray.cobol.loader.report( self.dde4 )
    def test_should_parse( self ):
        dde4= self.dde4
        self.assertEqual( 2920, dde4.size )
        self.assertEqual( 0, dde4.offset )
        self.assertEqual( 10, dde4.get("EMPLOYEE-TABLE" ).occurs.number(None) )
        self.assertEqual( 52, dde4.get("WEEK-RECORD" ).occurs.number(None) )
        self.assertEqual( 5, dde4.get("WEEK-RECORD" ).size )
        self.assertEqual( 52*5+32, dde4.get("EMPLOYEE-TABLE" ).size )
        self.assertEqual( "999999", dde4.get("EMPLOYEE-NO").sizeScalePrecision.final  )
        schema = stingray.cobol.loader.make_schema( [dde4] )
        schema_dict= dict( (a.name, a) for a in schema )
        self.assertEqual( (52*5+32)+32+5+4, schema_dict["LATE-ARRIVALS"].index(1,1).offset )
        self.assertEqual( (52*5+32)+32+5+5+4, schema_dict["LATE-ARRIVALS"].index(1,2).offset )
```

DDE Test copybook 5 from page 195 with simple redefines

Here is a redefines example.

```
page195= """
       01  REDEFINES-RECORD.
           05  A PICTURE X(6).
           05  B REDEFINES A.
               10  B-1 PICTURE X(2).
               10  B-2 PICTURE 9(4).
           05  C PICTURE 99V99.
"""
```

Be sure it parses. Be sure we can extract data.

```
class Test_Copybook_5( DDE_Test ):
    def setUp( self ):
        super().setUp()
        self.dde5= list(self.rf.makeRecord( self.lexer.scan(page195) ))[0]
        #stingray.cobol.loader.report( self.dde5 )
    def test_should_parse( self ):
        dde5= self.dde5
        self.assertEqual( 10, dde5.size )
        self.assertEqual( 6, dde5.get("A").size )
        self.assertEqual( 0, dde5.get("A").offset )
        self.assertEqual( 6, dde5.get("B").size )
        self.assertEqual( 0, dde5.get("B").offset )
        self.assertEqual( 2, dde5.get("B-1").size )
        self.assertEqual( 0, dde5.get("B-1").offset )
        self.assertEqual( 4, dde5.get("B-2").size )
        self.assertEqual( 2, dde5.get("B-2").offset )
        self.assertEqual( "9999", dde5.get("B-2").sizeScalePrecision.final )
        self.assertEqual( 4, dde5.get("C").size )
        self.assertEqual( 6, dde5.get("C").offset )

    def test_should_extract( self ):
        schema = stingray.cobol.loader.make_schema( [self.dde5] )
        schema_dict= dict( (a.name, a) for a in schema )
        data= stingray.cobol.Character_File( name="",
            file_object= ["AB12345678",],
            schema=schema )

        row= next( data.sheet( "" ).rows() )
        #stingray.cobol.dump( schema, row )

        self.assertEqual( "AB1234", row.cell(schema_dict["A"]).to_str() )
        self.assertEqual( "AB1234", row.cell(schema_dict["B"]).to_str() )
        self.assertEqual( "AB", row.cell(schema_dict["B-1"]).to_str() )
        self.assertEqual( "1234", row.cell(schema_dict["B-2"]).to_str() )
        self.assertEqual( "56.78", row.cell(schema_dict["C"]).to_str() )
```

DDE Test copybook 6 from page 197 with another redefines

```
page197= """
       01  REDEFINES-RECORD.
           05 NAME-2.
              10 SALARY PICTURE XXX.
              10 SO-SEC-NO PICTURE X(9).
              10 MONTH PICTURE XX.
           05 NAME-1 REDEFINES NAME-2.
              10 WAGE PICTURE 999V999.
              10 EMP-NO PICTURE X(6).
              10 YEAR PICTURE XX.
"""
```

Be sure it parses. Be sure we can extract data.

```
class Test_Copybook_6( DDE_Test ):
    def setUp( self ):
        super().setUp()
        self.dde6= list(self.rf.makeRecord( self.lexer.scan(page197) ))[0]
        #stingray.cobol.loader.report( self.dde6 )
    def test_should_parse( self ):
        dde6= self.dde6
        self.assertEqual( 3, dde6.get("SALARY").size )
        self.assertEqual( 0, dde6.get("SALARY").offset )
        self.assertEqual( 9, dde6.get("SO-SEC-NO").size )
        self.assertEqual( 3, dde6.get("SO-SEC-NO").offset )
        self.assertEqual( 2, dde6.get("MONTH").size )
        self.assertEqual( 12, dde6.get("MONTH").offset )
        self.assertEqual( 6, dde6.get("WAGE").size )
        self.assertEqual( 0, dde6.get("WAGE").offset )
        self.assertEqual( "999999", dde6.get("WAGE").sizeScalePrecision.final )
        self.assertEqual( 3, dde6.get("WAGE").usage.precision )
        self.assertEqual( 6, dde6.get("EMP-NO").size )
        self.assertEqual( 6, dde6.get("EMP-NO").offset )
        self.assertEqual( 2, dde6.get("YEAR").size )
        self.assertEqual( 12, dde6.get("YEAR").offset )

    def test_should_extract_1( self ):
        schema = stingray.cobol.loader.make_schema( [self.dde6] )
        schema_dict= dict( (a.name, a) for a in schema )
        data1= stingray.cobol.Character_File( name="",
            file_object= ["ABC123456789DE",],
            schema=schema )
        row= next( data1.sheet( "" ).rows() )
        #stingray.cobol.dump( schema, row )

        self.assertEqual( "ABC", row.cell(schema_dict["SALARY"]).to_str() )
        self.assertEqual( "123456789", row.cell(schema_dict["SO-SEC-NO"]).to_str() )
        self.assertEqual( "DE", row.cell(schema_dict["MONTH"]).to_str() )

    def test_should_extract_2( self ):
        schema = stingray.cobol.loader.make_schema( [self.dde6] )
        schema_dict= dict( (a.name, a) for a in schema )
        data2= stingray.cobol.Character_File( name="",
            file_object= ["123456ABCDEF78",],
            schema=schema )
        row= next( data2.sheet( "" ).rows() )
        #stingray.cobol.dump( schema, row )
        self.assertAlmostEquals( 123.456, row.cell(schema_dict["WAGE"]).to_float() )
        self.assertEqual( "ABCDEF", row.cell(schema_dict["EMP-NO"]).to_str() )
        self.assertEqual( "78", row.cell(schema_dict["YEAR"]).to_str() )
```

DDE Test copybook 7 from page 198, example "A"

```
page198A= """
       01  REDEFINES-RECORD.
           05 REGULAR-EMPLOYEE.
              10 LOCATION PICTURE A(8).
              10 GRADE PICTURE X(4).
              10 SEMI-MONTHLY-PAY PICTURE 9999V99.
              10 WEEKLY-PAY REDEFINES SEMI-MONTHLY-PAY
                  PICTURE 999V999.
           05 TEMPORARY-EMPLOYEE REDEFINES REGULAR-EMPLOYEE.
              10 LOCATION PICTURE A(8).
              10 FILLER PICTURE X(6).
              10 HOURLY-PAY PICTURE 99V99.
"""
```

Be sure it parses. Be sure we can extract data.

```
class Test_Copybook_7( DDE_Test ):
    def setUp( self ):
        super().setUp()
        self.dde7= list(self.rf.makeRecord( self.lexer.scan(page198A) ))[0]
        #stingray.cobol.loader.report( self.dde7 )
    def test_should_parse( self ):
        dde7= self.dde7
        self.assertEqual( 18, dde7.get("REGULAR-EMPLOYEE").size )
        self.assertEqual( 18, dde7.get("TEMPORARY-EMPLOYEE").size )
        self.assertEqual( 6, dde7.get("SEMI-MONTHLY-PAY").size )
        self.assertEqual( 6, dde7.get("WEEKLY-PAY").size )

    def test_should_extract_1( self ):
        schema = stingray.cobol.loader.make_schema( [self.dde7] )
        schema_dict= dict( (a.name, a) for a in schema )
        data1= stingray.cobol.Character_File( name="",
            file_object= ["ABCDEFGHijkl123456",],
            schema=schema )
        row= next( data1.sheet( "" ).rows() )
        # Can't dump with TEMPORARY-EMPLOYEE
        #stingray.cobol.dump( schema, row )

        self.assertEqual( '1234.56', row.cell(schema_dict["SEMI-MONTHLY-PAY"]).to_str() )

    def test_should_extract_2( self ):
        schema = stingray.cobol.loader.make_schema( [self.dde7] )
        schema_dict= dict( (a.name, a) for a in schema )
        data2= stingray.cobol.Character_File( name="",
            file_object= ["ABCDEFGHijklmn1234",],
            schema=schema )
        row= next( data2.sheet( "" ).rows() )
        # Can't dump with REGULAR-EMPLOYEE
        #stingray.cobol.dump( schema, row )

        self.assertEqual( '12.34', row.cell(schema_dict["HOURLY-PAY"]).to_str() )
```

DDE Test copybook 8 from page 198, example “B”

```
page198B= """
       01  REDEFINES-RECORD.
           05 REGULAR-EMPLOYEE.
               10 LOCATION PICTURE A(8).
               10 GRADE PICTURE X(4).
               10 SEMI-MONTHLY-PAY PICTURE 999V999.
           05 TEMPORARY-EMPLOYEE REDEFINES REGULAR-EMPLOYEE.
               10 LOCATION PICTURE A(8).
               10 FILLER PICTURE X(6).
               10 HOURLY-PAY PICTURE 99V99.
               10 CODE-H REDEFINES HOURLY-PAY PICTURE 9999.
"""
```

Be sure it parses. Be sure we can extract data.

```
class Test_Copybook_8( DDE_Test ):
    def setUp( self ):
        super().setUp()
        self.dde8= list(self.rf.makeRecord( self.lexer.scan(page198B) ))[0]
    def test_should_parse( self ):
        #stingray.cobol.loader.report( self.dde8 )
        dde8= self.dde8
        self.assertEqual( 18, dde8.get("REGULAR-EMPLOYEE").size )
        self.assertEqual( 18, dde8.get("TEMPORARY-EMPLOYEE").size )
        self.assertEqual( 6, dde8.get("SEMI-MONTHLY-PAY").size )
        self.assertEqual( 4, dde8.get("HOURLY-PAY").size )
        self.assertEqual( 4, dde8.get("CODE-H").size )

    def test_should_extract_1( self ):
        schema = stingray.cobol.loader.make_schema( [self.dde8] )
        schema_path_dict= dict( (a.path, a) for a in schema )
        data1= stingray.cobol.Character_File( name="",
            file_object= ["ABCDEFGHijkl123456",],
            schema=schema )

        row= next( data1.sheet( "" ).rows() )
        #stingray.cobol.dump( schema, row )
        #print( "SEMI-MONTHLY-PAY", schema_path_dict['REDEFINES-RECORD.REGULAR-EMPLOYEE.SEMI-MONTHLY-
        #print( "row.cell(...)", row.cell(schema_path_dict['REDEFINES-RECORD.REGULAR-EMPLOYEE.SEMI-MO
        self.assertAlmostEquals( 123.456,
            row.cell(schema_path_dict['REDEFINES-RECORD.REGULAR-EMPLOYEE.SEMI-MONTHLY-PAY']).to_float
        )

    def test_should_extract_2( self ):
        schema = stingray.cobol.loader.make_schema( [self.dde8] )
        schema_path_dict= dict( (a.path, a) for a in schema )
        data2= stingray.cobol.Character_File( name="",
            file_object= ["ABCDEFGHijklmn1234",],
            schema=schema )

        row= next( data2.sheet( "" ).rows() )
        #stingray.cobol.dump( schema, row )
        self.assertEqual( 12.34,
            row.cell(schema_path_dict['REDEFINES-RECORD.TEMPORARY-EMPLOYEE.HOURLY-PAY']).to_float()
        )
        self.assertEqual( 1234,
            row.cell(schema_path_dict['REDEFINES-RECORD.TEMPORARY-EMPLOYEE.CODE-H']).to_int()
        )

        schema_name_dict= dict( (a.name, a) for a in schema )
        self.assertEqual( "REDEFINES-RECORD.TEMPORARY-EMPLOYEE.HOURLY-PAY",
            schema_name_dict.get('HOURLY-PAY').path )
```

Test Copybook 9, Multiple 01 Levels

Some basic COBOL with two top-level records.

```
copy9= """
       01  DETAIL-LINE.
           05  QUESTION                    PIC ZZ.
           05  PRINT-YES                   PIC ZZ.
           05  PRINT-NO                    PIC ZZ.
           05  NOT-SURE                    PIC ZZ.
       01  SUMMARY-LINE REDEFINES DETAIL-LINE.
           05  COUNT                       PIC ZZ.
           05  FILLER                      PIC XX.
           05  FILLER                      PIC XX.
           05  FILLER                      PIC XX.
"""
```

Be sure it parses. Be sure we can extract data.

```
class Test_Copybook_9( DDE_Test ):
    def setUp( self ):
        super().setUp()
        self.dde9a, self.dde9b = self.rf.makeRecord( self.lexer.scan(copy9) )
        #stingray.cobol.loader.report( self.dde9a )
        #stingray.cobol.loader.report( self.dde9b )
    def test_should_parse( self ):
        dde9= self.dde9a
        self.assertEqual( 0, dde9.get( "QUESTION" ).offset )
        self.assertEqual( 2, dde9.get( "QUESTION" ).size )
        self.assertEqual( "ZZ", dde9.get( "QUESTION" ).sizeScalePrecision.final  )
        self.assertEqual( "", dde9.get( "QUESTION" ).usage.source() )
        self.assertEqual( 2, dde9.get( "PRINT-YES" ).offset )
        self.assertEqual( 2, dde9.get( "PRINT-YES" ).size )
        self.assertEqual( "ZZ", dde9.get( "PRINT-YES" ).sizeScalePrecision.final  )
        self.assertEqual( 4, dde9.get( "PRINT-NO" ).offset )
        self.assertEqual( 2, dde9.get( "PRINT-NO" ).size )
        self.assertEqual( "ZZ", dde9.get( "PRINT-NO" ).sizeScalePrecision.final  )
        self.assertEqual( 6, dde9.get( "NOT-SURE" ).offset )
        self.assertEqual( 2, dde9.get( "NOT-SURE" ).size )
        self.assertEqual( "ZZ", dde9.get( "NOT-SURE" ).sizeScalePrecision.final  )
        dde9= self.dde9b
        self.assertEqual( 0, dde9.get( "COUNT" ).offset )
        self.assertEqual( 2, dde9.get( "COUNT" ).size )
        self.assertEqual( "ZZ", dde9.get( "COUNT" ).sizeScalePrecision.final  )
        self.assertEqual( "", dde9.get( "COUNT" ).usage.source() )
    def test_should_extract( self ):
        schema = stingray.cobol.loader.make_schema( [self.dde9a, self.dde9b] )
        #print( schema )
        schema_dict= dict( (a.name, a) for a in schema )
        data= stingray.cobol.Character_File( name="",
            file_object= ["01020304",],
            schema=schema )

        row= next( data.sheet( "" ).rows() )
        #stingray..dump( schema, row )
        self.assertEqual( "1", row.cell(schema_dict['QUESTION']).to_str() )
        self.assertEqual( 2, row.cell(schema_dict['PRINT-YES']).to_int() )
        self.assertEqual( 3, row.cell(schema_dict['PRINT-NO']).to_float() )
        self.assertEqual( decimal.Decimal('4'), row.cell(schema_dict['NOT-SURE']).to_decimal() )
        self.assertEqual( "1", row.cell(schema_dict['COUNT']).to_str() )
```

Test Copybook 10, Occurs Depending On

The basic ODO situation: size depends on another item in the record.

```
copy10= """
       01  MAIN-AREA.
           03 REC-1.
             05 FIELD-1                       PIC 9.
             05 FIELD-2 OCCURS 1 TO 5 TIMES
                  DEPENDING ON FIELD-1        PIC X(05).
"""
```

Be sure it parses.

To be sure we can compute the offset, we need to extract data. For that, we'll need a mock stingray.cobol.COBOL_File to provide data for setting size and offset.

```
class Test_Copybook_10( DDE_Test ):
    def setUp( self ):
        super().setUp()
        self.dde10 = list(self.rf.makeRecord( self.lexer.scan(copy10) ))[0]
        #stingray.cobol.loader.report( self.dde10 )
    def test_should_parse( self ):
        dde10= self.dde10

        self.assertEqual( 0, dde10.get( "FIELD-1" ).offset )
        self.assertEqual( 1, dde10.get( "FIELD-1" ).size )
        self.assertEqual( "9", dde10.get( "FIELD-1" ).sizeScalePrecision.final  )
        self.assertEqual( "", dde10.get( "FIELD-1" ).usage.source() )

        self.assertEqual( 0, dde10.get( "FIELD-2" ).offset )
        self.assertEqual( 5, dde10.get( "FIELD-2" ).size )
        self.assertEqual( "XXXXX", dde10.get( "FIELD-2" ).sizeScalePrecision.final  )
        self.assertEqual( "", dde10.get( "FIELD-2" ).usage.source() )

    def test_should_setsizeandoffset( self ):
        dde10= self.dde10

        schema= stingray.cobol.loader.make_schema( [dde10] )
        self.data = stingray.cobol.Character_File( name="",
            file_object= ["3111112222233333",],
            schema=schema )
        row= next( self.data.sheet( "IGNORED" ).rows() )

        self.assertEqual( 0, dde10.get( "FIELD-1" ).offset )
        self.assertEqual( 1, dde10.get( "FIELD-1" ).size )
        self.assertEqual( "9", dde10.get( "FIELD-1" ).sizeScalePrecision.final  )
        self.assertEqual( "", dde10.get( "FIELD-1" ).usage.source() )
        self.assertEqual( 1, dde10.get( "FIELD-2" ).offset )
        self.assertEqual( 5, dde10.get( "FIELD-2" ).size )
        self.assertEqual( "XXXXX", dde10.get( "FIELD-2" ).sizeScalePrecision.final  )
        self.assertEqual( "", dde10.get( "FIELD-2" ).usage.source() )
```

Test Copybook 11, Complex Occurs Depending On

A fairly complex ODO situation: size and offset depends other items in the record.

```
copy11= """
       01  MAIN-AREA.
           03 REC-1.
              05 FIELD-1                       PIC 9.
              05 FIELD-3                       PIC 9.
              05 FIELD-2 OCCURS 1 TO 5 TIMES
                   DEPENDING ON FIELD-1        PIC X(05).
           03 REC-2.
              05 FIELD-4 OCCURS 1 TO 5 TIMES
                   DEPENDING ON FIELD-3        PIC X(05).
"""
```

Be sure it parses.

To be sure we can compute the offset, we need to extract data. For that, we'll need a mock `stingray.cobol.COBOL_File` to provide data for setting size and offset.

```
class Test_Copybook_11( DDE_Test ):
    def setUp( self ):
        super().setUp()
        self.dde11 = list(self.rf.makeRecord( self.lexer.scan(copy11) ))[0]
        #stingray.cobol.loader.report( self.dde11 )

    def test_should_parse( self ):
        dde11= self.dde11

        self.assertEqual( 0, dde11.get( "FIELD-1" ).offset )
        self.assertEqual( 1, dde11.get( "FIELD-1" ).size )
        self.assertEqual( "9", dde11.get( "FIELD-1" ).sizeScalePrecision.final  )
        self.assertEqual( "", dde11.get( "FIELD-1" ).usage.source() )

        self.assertEqual( 0, dde11.get( "FIELD-2" ).offset )
        self.assertEqual( 5, dde11.get( "FIELD-2" ).size )
        self.assertEqual( "XXXXX", dde11.get( "FIELD-2" ).sizeScalePrecision.final  )
        self.assertEqual( "", dde11.get( "FIELD-2" ).usage.source() )

        self.assertEqual( 0, dde11.get( "FIELD-3" ).offset )
        self.assertEqual( 1, dde11.get( "FIELD-3" ).size )
        self.assertEqual( "9", dde11.get( "FIELD-3" ).sizeScalePrecision.final  )
        self.assertEqual( "", dde11.get( "FIELD-3" ).usage.source() )

        self.assertEqual( 0, dde11.get( "FIELD-4" ).offset )
        self.assertEqual( 5, dde11.get( "FIELD-4" ).size )
        self.assertEqual( "XXXXX", dde11.get( "FIELD-4" ).sizeScalePrecision.final  )
        self.assertEqual( "", dde11.get( "FIELD-4" ).usage.source() )

    def test_should_setsizeandoffset( self ):
        dde11= self.dde11
        schema= stingray.cobol.loader.make_schema( [dde11] )
        self.data = stingray.cobol.Character_File( name="",
            file_object= ["321111122222333334444455555",],
            schema=schema )
        row= next( self.data.sheet( "" ).rows() )

        self.assertEqual( 0, dde11.get( "FIELD-1" ).offset )
        self.assertEqual( 1, dde11.get( "FIELD-1" ).size )
        self.assertEqual( "9", dde11.get( "FIELD-1" ).sizeScalePrecision.final  )
        self.assertEqual( "", dde11.get( "FIELD-1" ).usage.source() )

        self.assertEqual( 2, dde11.get( "FIELD-2" ).offset )
        self.assertEqual( 5, dde11.get( "FIELD-2" ).size )
        self.assertEqual( "XXXXX", dde11.get( "FIELD-2" ).sizeScalePrecision.final  )
        self.assertEqual( "", dde11.get( "FIELD-2" ).usage.source() )

        self.assertEqual( 1, dde11.get( "FIELD-3" ).offset )
        self.assertEqual( 1, dde11.get( "FIELD-3" ).size )
        self.assertEqual( "9", dde11.get( "FIELD-3" ).sizeScalePrecision.final  )
        self.assertEqual( "", dde11.get( "FIELD-3" ).usage.source() )

        self.assertEqual( 17, dde11.get( "FIELD-4" ).offset )
        self.assertEqual( 5, dde11.get( "FIELD-4" ).size )
        self.assertEqual( "XXXXX", dde11.get( "FIELD-4" ).sizeScalePrecision.final  )
        self.assertEqual( "", dde11.get( "FIELD-4" ).usage.source() )
```

Test Suite and Runner

In case we want to build up a larger test suite, we avoid doing any real work unless this is the main module being executed.

```
import test
suite= test.suite_maker( globals() )

if __name__ == "__main__":
    with test.Logger( stream=sys.stdout, level=logging.INFO ):
        logging.getLogger( "stingray.cobol.defs" ).setLevel( logging.DEBUG )
        logging.info( __file__ )
        #unittest.TextTestRunner().run(suite())
        unittest.main( Test_Copybook_11() ) # Specific debugging
```

1.14.10 Test Snappy and Protobuf

These are tests for the snappy decompression and the protobuf object encoding.

Overheads

```
"""stingray.snappy and stingray.protobuf Unit Tests."""
import unittest
import io
import stingray.snappy
import stingray.protobuf
import logging, sys
```

Snappy Reader

The snappy decompressor is built around several layers of protocol.

1. There’s a framing protocol with a frame type and size.
2. There’s a size for the uncompressed data in the frame.
3. There are the 4 kinds of snappy tags that create the data.
    - 00 : literal
    - 01, 10, 11 : copies with various kinds of sizes and offsets.

```
class Test_Snappy( unittest.TestCase ):
    def setUp( self ):
        #logging.getLogger().setLevel( logging.DEBUG )
        self.buffer = io.BytesIO(
            b’\x00\x11\x00\x00’ # Header: 17 bytes in frame
            b’\x18’ # Size of the uncompressed data
            b’\x14hi mom’ # 000101,00 -> literal of 5+1=6 bytes
            b’\x09\x06’ # 000,010,01 00000110 -> copy 4+2=6 bytes offset of 6
            b’\x16\x00\x0C’ # 000101,10 00000000 00001100 -> copy 5+1=6 bytes offset of 12
            b’\x17\x00\x00\x12’ # 000101,11 00000000 00000000 00000000 00010010 -> copy 5+1=6 bytes

            b’\x00\x0a\x00\x00’ # Header: 10 bytes in frame
            b’\x08’ # Size of the uncompressed data
            b’\x1CZYXW!@#$’ # 000111,00 -> literal of 7+1=8 bytes
            )
        self.snappy= stingray.snappy.Snappy()
    def test_should_decompress( self ):
        data= self.snappy.decompress( self.buffer )
        self.assertEqual( b’hi momhi momhi momhi momZYXW!@#$’, data )
```

Protobuf Decoder

The protobuf decoder unpacks various kinds of data to create simple Message instances.

This is a two-layer protocol.

There’s a protobuf-encoded ArchiveInfo message.

```
message ArchiveInfo {
    optional uint64 identifier = 1;
    repeated MessageInfo message_infos = 2;
}
```

This contains a protobuf-encoded MessageInfo message.

```
message MessageInfo {
    required uint32 type = 1;
    repeated uint32 version = 2 [packed = true];
    required uint32 length = 3;
    repeated FieldInfo field_infos = 4;
    repeated uint64 object_references = 5 [packed = true];
    repeated uint64 data_references = 6 [packed = true];
}
```

The message has a payload, which is the relevant message that is part of the Numbers workbook.

Here’s the proto definition for the test message encoded in the payload below.

```
message DocumentArchive {
  repeated .TSP.Reference sheets = 1;
  required .TSA.DocumentArchive super = 8;
  optional .TSP.Reference calculation_engine = 3 [deprecated = true];
  required .TSP.Reference stylesheet = 4;
  required .TSP.Reference sidebar_order = 5;
  required .TSP.Reference theme = 6;
  optional .TN.UIStateArchive uistate = 7;
  optional .TSP.Reference custom_format_list = 9;
  optional string printer_id = 10;
  optional string paper_id = 11;
  optional .TSP.Size page_size = 12;
}
```

Note that each part of this (except for `printer_id` and `paper_id`) could involve decoding a contained message. We don't recursively descend, however, merely decoding the top message.

```
class Test_Protobuf( unittest.TestCase ):
    def setUp( self ):
        #logging.getLogger("Archive_Reader").setLevel( logging.DEBUG )
        self.buffer= bytes( (
            37, # 37 bytes follow.
            # AchiveInfo and MessageInfo
            8, 1, 18, 33, 8, 1, 18, 3, 1, 0, 5, 24, 91, 42, 22, 133, 12, 170, 15, 134,
            12, 185, 14, 132, 12, 137, 12, 135, 12, 183, 14, 136, 12, 184, 15, 183, 15,
            # Payload
            10, 3, 8, 183, 14, 10, 3, 8, 170, 15, 10, 3, 8, 183, 15, 34, 3, 8, 136, 12,
            42, 3, 8, 185, 14, 50, 3, 8, 137, 12, 66, 33, 10, 5, 58, 3, 8, 132, 12, 26,
            2, 101, 110, 34, 3, 8, 133, 12, 42, 3, 8, 184, 15, 50, 3, 8, 135, 12, 58,
            3, 8, 134, 12, 64, 0, 82, 1, 32, 90, 9, 110, 97, 45, 108, 101, 116, 116,
            101, 114, 98, 10, 13, 0, 0, 25, 68, 21, 0, 0, 70, 68,
            )
        )
        self.reader= stingray.protobuf.Archive_Reader()
    def test_should_decode( self ):
        message_list = list( self.reader.archive_iter(self.buffer) )
        m0_id, m0_m = message_list[0]
        self.assertEqual( 1, m0_id )
        self.assertEqual( "TN.DocumentArchive", m0_m.name_ )
        sheets= m0_m[1]
        self.assertEqual( [(8, 183, 14), (8, 170, 15), (8, 183, 15)], sheets )
        super= m0_m[8]
        self.assertEqual( [(10, 5, 58, 3, 8, 132, 12, 26, 2, 101, 110, 34, 3, 8, 133, 12, 42, 3, 8, 1
        calculation_engine= m0_m[3]
        self.assertEqual( [], calculation_engine )
        stylesheet= m0_m[4]
        self.assertEqual( [(8, 136, 12)], stylesheet )
        sidebar_order= m0_m[5]
        self.assertEqual( [(8, 185, 14)], sidebar_order )
        theme= m0_m[6]
        self.assertEqual( [(8, 137, 12)], theme )
        uistate= m0_m[7]
        self.assertEqual( [], uistate )
        custom_format_list= m0_m[9]
        self.assertEqual( [], custom_format_list )
        printer_id= m0_m[10]
        self.assertEqual( [(32,)], printer_id )
        paper_id= m0_m[11]
        # b'na-letter' is the paper_id
        self.assertEqual( [(110, 97, 45, 108, 101, 116, 116, 101, 114)], paper_id )
        page_size= m0_m[12]
        self.assertEqual( [(13, 0, 0, 25, 68, 21, 0, 0, 70, 68)], page_size )
```

Test Suite and Runner

In case we want to build up a larger test suite, we avoid doing any real work unless this is the main module being executed.

```
import test
suite= test.suite_maker( globals() )

if __name__ == "__main__":
    print( __file__ )
    unittest.TextTestRunner().run(suite())
```

1.14.11 stingray.test `__init__.py`

Overheads.

```
"""Stingray Schema-based File Reader Testing Package"""

import unittest
import logging, sys
```

This is a slightly modified version of the test discovery function in the Python `unittest` package.

```
def suite_maker( source ):
    def make_suite():
        tests= unittest.TestSuite()
        for name in source:
            obj = source[name]
            if isinstance(obj, type) and issubclass(obj, unittest.TestCase):
                tests.addTests(
                    unittest.defaultTestLoader.loadTestsFromTestCase(obj))
        return tests
    return make_suite
```

A handy logging context.

```
class Logger:
    def __init__( self, **kw ):
        self.args= kw
    def __enter__( self ):
        logging.basicConfig( **self.args )
    def __exit__( self, *exc ):
        logging.shutdown()
```

We can use `with test.Logger( stream=sys.stderr, level=logging.DEBUG ):` To enable logging.

1.15 Stingray Build

The source for `stingray` is a Sphinx project that depends on PyLit. Yes. The documentation spawns the code.

Important: PyLit Feature

The first line of each file should be `..    #!/usr/bin/env python3`.

The four spaces between `..` and `#!` defines the indent used for each line of code in the file.

Four spaces.

Stingray depends on the following

- xlrd. http://www.lexicon.net/sjmachin/xlrd.htm

  Version 0.9.2 is Python 3 compatible. https://pypi.python.org/pypi/xlrd/0.9.2

In addition to Python 3.3, there are two other projects used to build.

- PyLit. https://github.com/slott56/PyLit-3
- Sphinx. http://sphinx.pocoo.org/

The PyLit install is little more than a download and move the `pylit.py` file to the Python `site-packages` directory.

Sphinx and XLRD should be installed with easy_install.

```
easy_install xlrd
easy_install sphinx
```

In the case of having Python2 and Python3 installed, `easy_install-3.3` may be required. On most systems, `sudo` is also required.

The diagrams are done with YUML. See http://yuml.me.

1.15.1 Build Procedure

1. Bootstrap the `build.py` script by running PyLit.

   ```
   python3 -m pylit -t source/build.rst build.py
   ```

   This reports that an extract was written to `build.py`.

2. Use the `build.py` script to create the `stingray` source, unit tests, demonstration applications. Build the Sphinx documentation. And run the unit test, too.

   ```
   python3 build.py
   ```

   At the end of this step, the directory tree will include the following.

   - `build`. The documentation. In HTML.
   - `demo`. Some demonstration applications that use `stingray`. See *Stingray Demo Applications*.
   - `stingray`. The Python library, ready for installation.
   - `test`. The unit test script.

   This reports, also, that 174 tests were run.

In general (i.e., any OS except Windows), it's sensible to do this:

```
chmod +x build.py
```

This allows us to use the following for a rebuild:

```
./build.py
```

1.15.2 Build Script Design

This is a platform-independent `build.py` file for the build script. This can use `from sphinx.application import Sphinx` and `import pylit` to access these modules from within Python instead of using command-line shell script techniques.

Overheads

We’re going to make use of three “applications”.

- Sphinx top-level application.
- PyLit top-level application.
- Unittest top-level test runner.

```
"""Platform-independent build script"""
from __future__ import print_function
import os
import sys
import errno
from sphinx.application import Sphinx
import pylit
import unittest
import logging
import shutil
```

Sphinx Build

`sphinx_build`(*srcdir*, *outdir*, *buildername=’html’*)

This function handles the simple use case for the `sphinx-build` script.

```
def sphinx_build( srcdir, outdir, buildername='html' ):
    """Essentially: ``sphinx-build $* -b html source build/html``"""
    confdir= srcdir= os.path.abspath( srcdir )
    outdir= os.path.abspath( outdir )
    doctreedir = os.path.join(outdir, '.doctrees')
    app = Sphinx(srcdir, confdir, outdir, doctreedir, buildername)
    app.build(force_all=False, filenames=[])
    return app.statuscode
```

PyLit Build

`pylit_build`(*srcdir*, *outdir*)

This function handles the simple use case for PyLit.

This also handles the necessary rewrite to modify standard paths to Windows paths.

```
def pylit_build( infile, outfile ):
    """Essentially: ``python3 -m pylit -t source/demo/data_quality.rst demo/test.py``
    The issue here is that we need to provide platform-specific paths.
    """
    if os.sep != '/':
        # Fix windows paths.
        infile= os.path.join( *infile.split('/') )
        outfile= os.path.join( *outfile.split('/') )
    pylit.main( txt2code= True, overwrite="yes", infile= infile, outfile= outfile )
```

Make Directories

mkdir (*path*)

This function handles the simple use case for assuring that the directory tree exists.

This also handles a rewrite to modify standard paths to Windows paths.

```
def mkdir( path ):
    if os.sep != '/':
        # Fix windows paths.
        path= os.path.join( *path.split('/') )
    try:
        os.makedirs( path )
    except OSError as e:
        if e.errno == errno.EEXIST:
            pass
        else:
            raise
```

Copy Data File

copy_file (*srcdir*, *outdir*)

This function handles the simple use case for copying a file

```
def copy_file( srcdir, outdir ):
    """Essentially: ``cp srcdir outdir``"""
    shutil.copy2( srcdir, outdir )
```

Run the Test Script

run_test ()

In effect, this does `python3 test/main.py`

```
def run_test():
    from test.main import suite
    from test import Logger
    with Logger( stream=sys.stdout, level=logging.WARN ):
        unittest.TextTestRunner().run(suite())
```

The Build Sequence

```
def build():
    mkdir( 'stingray/schema' )
    mkdir( 'stingray/cobol' )
    mkdir( 'stingray/workbook' )

    pylit_build( 'source/stingray_init.rst', 'stingray/__init__.py' )
    pylit_build( 'source/cell.rst', 'stingray/cell.py' )
    pylit_build( 'source/sheet.rst', 'stingray/sheet.py' )
    pylit_build( 'source/workbook/init.rst', 'stingray/workbook/__init__.py' )
    pylit_build( 'source/workbook/base.rst', 'stingray/workbook/base.py' )
    pylit_build( 'source/workbook/csv.rst', 'stingray/workbook/csv.py' )
    pylit_build( 'source/workbook/xls.rst', 'stingray/workbook/xls.py' )
    pylit_build( 'source/workbook/xlsx.rst', 'stingray/workbook/xlsx.py' )
    pylit_build( 'source/workbook/ods.rst', 'stingray/workbook/ods.py' )
    pylit_build( 'source/workbook/numbers_09.rst', 'stingray/workbook/numbers_09.py' )
    pylit_build( 'source/workbook/numbers_13.rst', 'stingray/workbook/numbers_13.py' )
    pylit_build( 'source/workbook/fixed.rst', 'stingray/workbook/fixed.py' )
    pylit_build( 'source/schema.rst', 'stingray/schema/__init__.py' )
    pylit_build( 'source/schema_loader.rst', 'stingray/schema/loader.py' )
    pylit_build( 'source/cobol_init.rst', 'stingray/cobol/__init__.py' )
    pylit_build( 'source/cobol_loader.rst', 'stingray/cobol/loader.py' )
    pylit_build( 'source/cobol_defs.rst', 'stingray/cobol/defs.py' )
    pylit_build( 'source/snappy.rst', 'stingray/snappy.py' )
    pylit_build( 'source/protobuf.rst', 'stingray/protobuf.py' )
    pylit_build( 'source/installation.rst', 'setup.py' )

    copy_file( 'source/Numbers.json', 'stingray/Numbers.json' )
    copy_file( 'source/Common.json', 'stingray/Common.json' )

    mkdir( 'test' )

    pylit_build( 'source/testing/test_init.rst', 'test/__init__.py' )
    pylit_build( 'source/testing/main.rst', 'test/main.py' )
    pylit_build( 'source/testing/cell.rst', 'test/cell.py' )
    pylit_build( 'source/testing/sheet.rst', 'test/sheet.py' )
    pylit_build( 'source/testing/schema.rst', 'test/schema.py' )
    pylit_build( 'source/testing/schema_loader.rst', 'test/schema_loader.py' )
    pylit_build( 'source/testing/workbook.rst', 'test/workbook.py' )
    pylit_build( 'source/testing/cobol.rst', 'test/cobol.py' )
    pylit_build( 'source/testing/cobol_loader.rst', 'test/cobol_loader.py' )
    pylit_build( 'source/testing/cobol_2.rst', 'test/cobol_2.py' )
    pylit_build( 'source/testing/snappy_protobuf.rst', 'test/snappy_protobuf.py' )

    mkdir( 'demo' )

    pylit_build( 'source/demo/data_quality.rst', 'demo/test.py' )
    pylit_build( 'source/demo/validation.rst', 'demo/app.py' )
    pylit_build( 'source/demo/profile.rst', 'demo/profile.py' )
    pylit_build( 'source/demo/cobol_reader.rst', 'demo/cobol_reader.py' )

    run_test()

    sphinx_build( 'source', 'build/html', 'html' )
    sphinx_build( 'source', 'build/latex', 'latex' )
```

Main Program Switch

When the build.py script is run from the command line, it will execute the build() function. When it is imported, however, it will do nothing special.

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

1.15.3 Additional Builds

Sometimes it's desriable to refresh the documentation.

The HTML pages are built with this command.

```
sphinx-build $* -b html source build/html
```

The LaTeX document is built with this command.

```
sphinx-build $* -b latex source build/latex
```

1.16 Installation via setup.py

Use the following link to get the latest code:

> git clone git://git.code.sf.net/p/stingrayreader/code
> stingrayreader-code

It's also possible to get an archive distribution kit. These may be slightly out of date.

1.16.1 Optional Build-from-Scratch

The Stingray distribution kit (minimally) is just the following.

- source. The RST-formatted source used by PyLit3 to to create code and documentation.
- sample. Several sample workbooks used for testing.
- build.py. The build procedure.

Given these two directories, the build procedure uses PyLit3 to create stingray package. It uses Sphinx to create the documentation.

See *Stingray Build* for more information on the build procedure. This is an optional step to recreate the Python code from the documentation.

The build process required PyLit3 and Sphinx to be installed. Setuptools can be used to simplify installation.

- PyLit. https://github.com/slott56/PyLit-3
- Sphinx. http://sphinx.pocoo.org/
- Setuptools. http://pypi.python.org/pypi/setuptools

```
python3 build.py
```

1.16.2 Installation via Distutils

To install `stingray` you can use the following. On Linux, this may require privileges via `sudo`.

```
sudo python3 setup.py install
```

In some cases, you might want to break this down into a build step that doesn't require privileges and the final install step, which does require privileges.

```
python3 setup.py build
sudo python3 setup.py install
```

1.16.3 The `setup.py` File

To use Stingray, you must have the following package installed.

- xlrd. http://www.lexicon.net/sjmachin/xlrd.htm

  Version 0.9.2 is Python 3 compatible. https://pypi.python.org/pypi/xlrd/0.9.2

This provides Python package information.

```
from setuptools import setup
import sys

if sys.version_info < (3,3):
    print( "Stringray requires Python 3.3" )
    sys.exit(2)

setup(
    name='stingray',
    version='4.4.2',
    description='Schema-Based File Reader, COBOL, EBCDIC Conversion, ETL, Data Profiling',
    author='S.Lott',
    author_email='s_lott@yahoo.com',
    url='https://sourceforge.net/projects/stingray/',
    packages=[
        'stingray',
        'stingray.schema',
        'stingray.cobol',
        'stingray.workbook',
        ],
    package_data={'stingray': ['*.json']},
```

We depend on the release of Python itself, based on the `sys.version_info` named tuple. We have a few Python3.3 features.

We depend XLRD, which comes from PyPi. This is required for normal use and operation.

For a full build from the source document, we depend on having a new release of Sphinx and PyLit3.

The `obsoletes` is there in case anyone did happen to download the previous release.

```
install_requires=["xlrd>=0.9","sphinx>1.2"],
obsoletes=["cobol_dde","data_profile"],
```

Here are some trove classifiers.

```
classifiers=[
    "Development Status :: 6 - Mature",
    "Environment :: Console",
    "Intended Audience :: Developers",
    "Operating System :: OS Independent",
    "Programming Language :: Python",
    "Programming Language :: COBOL",
    "Topic :: Database",
    "Topic :: Software Development :: Libraries",
    "Topic :: Software Development :: Quality Assurance",
    ],
)
```

CHAPTER

TWO

LICENSE

CHAPTER

THREE

THE TODO LIST

Todo

Test hashable interface of Cell

(The *original entry* is located in /Users/slott/Documents/Projects/Stingray-4.4/source/cell.rst, line 214.)

Todo

refactor setSizeAndOffset()

Refactor setSizeAndOffset() into the `Allocation` class methods to remove isinstance() nonsense.

(The *original entry* is located in /Users/slott/Documents/Projects/Stingray-4.4/source/cobol_defs.rst, line 1168.)

Todo

Fix performance.

This is called once per row: it needs to be simpler and faster. Some refactoring can eliminate the if statements.

(The *original entry* is located in /Users/slott/Documents/Projects/Stingray-4.4/source/cobol_defs.rst, line 1180.)

Todo

88-level items could create boolean-valued properties.

(The *original entry* is located in /Users/slott/Documents/Projects/Stingray-4.4/source/cobol_loader.rst, line 397.)

Todo

Index by name and path, also.

This will eliminate some complexity in COBOL schema handling where we create the a "schema dictionary" using simple names and path names.

(The *original entry* is located in /Users/slott/Documents/Projects/Stingray-4.4/source/schema.rst, line 334.)

Todo

EBCDIC File V format with Occurs Depending On to show the combination.

(The *original entry* is located in /Users/slott/Documents/Projects/Stingray-4.4/source/testing/cobol.rst, line 472.)

Todo

Test EXTERNAL, GLOBAL as Skipped Words, too.

(The *original entry* is located in /Users/slott/Documents/Projects/Stingray-4.4/source/testing/cobol_loader.rst, line 990.)

Todo

Refactor workbook package

This module needs to be rebuilt into a package which imports a number of subsidiary modules. It's too large as written.

Adding Numbers '13 will make this module even more monstrous. Adding future spreadsheets will on exacerbate the problem.

It should become (like `cobol`) a high-level package that imports top-level classes from modules within the package.

> `from workbook.csv import CSV_Workbook`
>
> `from workbook.xls import XLS_Workbook`
>
> *... etc. ...*

This should make a transparent change from module to package.

The top-level definition for `cobol.Workbook` must to be refactored into a `base` module that can be shared by all the modules in the package that extend this base definition.

(The *original entry* is located in /Users/slott/Documents/Projects/Stingray-4.4/source/workbook/init.rst, line 170.)

Todo

Additional Numbers13_Workbook Feature

Translate Formula and Formula error to Text

(The *original entry* is located in /Users/slott/Documents/Projects/Stingray-4.4/source/workbook/numbers_13.rst, line 27.)

Todo

Refactor this, it feels clunky.

(The *original entry* is located in /Users/slott/Documents/Projects/Stingray-4.4/source/workbook/ods.rst, line 120.)

CHAPTER

FOUR

INDICES AND TABLES

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