Summary/ Cheat-sheet:

Data Reading:

```
data = B[5,5]                 # return data point
data = B[0, 0:20]             # returns array of your data type
data = B["A1:AA100"]          # returns generator for a matrix of data
data = B[0:20:2, 0:40:6]      # stepping/splicing.  Don't use in Excel
data = B[:20, (0,5,8)]        # select specific points to retrieve (advanced stepping)
data = B[0, 0:]               # INVALID INPUT.  Length needs to be specified
```

Data Writing:

```
B[0,0] = 4      # single point
B[0, :20] = 44            # copying 44 over a row.  Splice specifies length
B[:20, 0] = 44            # copying 44 over a column.  Splice specifies length
B[0, 0:] = 44             # INVALID INPUT.  Length needs to be specified
B[1, 0:] = range(100)         # writes out an array.  Note that the splice only specifies the
                              # direction (does not specify the length), and can be left empty
B[0,0] = [[n*p for n in range(100)] for p in range(100)]     # writing a matrix
* Same addressing features as Data Reading.
```

Referencing:

```
B.change_cellref(0,5)         # makes it so that calling B[0,0] will return the value at 0,5
                              #   recommended that you do not use if you use string addressing
B.change_sheet('Sheet1')      # sheet addressing with string
B.change_sheet(1)             # sheet addressing with index
B.current_sheet_name          # stores current sheet name.  Do not change
B.change_workbook('Book1.xlsx')   # workbook addressing, string or int (use string)
B.current_workbook_name       # current workbook name.  Do not change
B.working_directory           # variable stores current workbook's working directory.  Only for
                              # Gnumeric right now
```

Sheet Addressing in Call:

```
data = B["A1:B3", 'Sheet3']   # gets the data from Sheet3 without needing to change sheet
```

Creating/Renaming Sheets:

```
B.create_sheet('sheetname')
```

```
B.rename_sheet('oldsheetname', 'newsheetname')
```

Data Types:

```
B.change_dtype(list)          # change dtype to list
B.change_dtype(tuple)
B.change_dtype(int)           # change dtype to array of numpy integers
B.change_dtype(float)         # change dtype to array of numpy floating point values
```

Sheet Write Protection:

```
B.protect_sheet('Sheet1')     # protects a sheet.  Not tested across workbooks.  May be buggy
B.unprotect_sheet('Sheet1')
```

More Advanced (and probably useless) features:

```
B.ALWAYS_GENERATE = True     #  arrays will be returned as generator-like objects
                             #  (matricies will still be generators of arrays)
```

To Come ** NOT YET IMPLEMENTED **:

```
B.make_workbook               # creates a workbook
B.open_workbook               # opens a workbook from a filename
B.close_workbook              # closes an open workbook
B.format_cells('A1:B7', formating)   # format a cell range
```

```
12.0, 15.0))
```
same with tuples

```
>>> B.change_dtype(int)
>>> data = B['A1: F20']
>>> data[:4]
array([[ 0, 0, 0, 0, 0, 0],
 [ 0, 1, 2, 3, 4, 5],
 [ 0, 2, 4, 6, 8, 10],
 [ 0, 3, 6, 9, 12, 15]])
```
same with numpy array

What are the advantages of this? Well, maybe I should talk about performance, but first I should mention one more thing quick for the advanced users

Note on Simultaneous Generators:

> In some applications, simultaneous generators are bad. However, you should be able to safely use them with any PyWorksheet class – across workbooks even. Be careful though, because they are generators, if you change the data it is going to read, then that could affect your output (essentially treat them as pointers to immutable data like a list or dictionary)

Performance:

I've worked hard to make my classes and return types operate as fast as possible, and dtypes are the biggest part of the work I've done. The code written for the data call routine (i.e. data[0], etc) is as fast as I could make it, with as little memory overhead as possible (I've tested it, comparing other alternatives). If you know how to make it faster, please tell me, but it is pretty quick. Just know that if you use the standard calls shown above, you will get data as fast as it can be generated (as far as I know). This could be important in your spreadsheet program, as you don't want your python taking up too much computing power or memory when you have several macros running at once!