Upcoming PyUNO improvements in LibreOffice 5.1

Matthew Francis
Introduction

- The state of PyUNO as of LibreOffice 5.0
  - Implemented near the dawn of Python, barely changed since
  - Very little syntactic sugar
  - Generally a lot like writing Java
  - Slower than it should be – especially from a remote process

- What I hope to achieve with the new changes for 5.1
  - Make working with UNO in Python feel more Pythonic
    - Less verbose – make use of available Python syntax
  - Make it faster than before
  - Personal goal – make PyUNO more appropriate as a base to build automated UI tests on
New features in PyUNO for 5.1
Collection interfaces

- Indexed array interfaces
  - `com::sun::star::container::XIndexAccess`
  - `com::sun::star::container::XIndexReplace`
  - `com::sun::star::container::XIndexContainer`

- What's changed?
  - UNO objects implementing these interfaces now behave like Python lists
Collection interfaces

Whenever you see method calls like the following, there is a simpler way to do it:

- `count = obj.getCount()`  ⇒  `count = len(obj)`
- `Value = obj.getByIndex(0)`  ⇒  `value = obj[0]`
- `obj.replaceByIndex(0, value)`  ⇒  `obj[0] = value`
- `obj.insertByIndex(0, value)`  ⇒  `obj[0:0] = value`
- `obj.removeByIndex(0)`  ⇒  `del obj[0]`

Iteration and testing value presence also works:

- `for value in obj: ...`
- `if value in obj: ...`
  - ...but the `if – in` syntax is probably useful only rarely for indexed collections (and not efficient)
Example: iterating over document footnotes – the old way

doc = ... # load a text document
count = doc.Footnotes.getCount()
for i in range(count):
    footnote = doc.Footnotes.getByIndex(i)
    print(footnote.String)
Example: iterating over document footnotes – the new way

doc = ... # load a text document
count = len(doc.Footnotes)
for i in range(len):
    print(doc.Footnotes[i].String)

Or even better, if the index isn't important:

doc = ... # load a text document
for footnote in doc.Footnotes:
    print(footnote.String)
Other examples of XIndex*
  - Text document
    - Redlines
    - Endnotes
  - Spreadsheet
    - Charts
    - NamedRanges
Collection interfaces

- Associative array interfaces
  - com::sun::star::container::XNameAccess
  - com::sun::star::container::XNameReplace
  - com::sun::star::container::XNameContainer

- What's changed?
  - UNO objects implementing these interfaces now behave like Python dicts
Collection interfaces

Whenever you see method calls like the following, there is a simpler way to do it:

- Value = obj.getName(key) ⇒ value = obj[key]
- if obj.hasByName(key): ... ⇒ if key in obj: ...
- obj.replaceByName(key, value) ⇒ obj[key] = value
- obj.insertByName(key, value) ⇒ obj[key] = value
- obj.removeByName(key) ⇒ del obj[key]

Iteration and testing value presence also works – for keys:

- for key in obj: ...
- if key in obj: ...
  - Different from indexed collections – the for and in operators tests for keys, not values
Example: Navigate elements of a spreadsheet – the old way

```python
spr = ... # load a spreadsheet
sheet = spr.Sheets.getByName('Sheet1')
range = sheet.NamedRanges.getByName('MyRange')
```
Collection interfaces

Example: Navigate elements of a spreadsheet – the new way

```
spr = ... # load a spreadsheet
sheet = spr.Sheets['Sheet1']
range = sheet.NamedRanges['MyRange']
```
Collection interfaces

What if an object supports both XIndex* and XName*?

You can access it using both `obj[0]` and `obj['Name']`

However, iterating yields keys rather than values

Like a Python dict

Examples:

- Text document
  - TextTables
  - EmbeddedObjects
  - GraphicObjects
Collection interfaces

- Enumerations
  - `com::sun::star::container::XEnumerationAccess`
  - `com::sun::star::container::XEnumeration`

- What's changed
  - You can iterate over UNO enumerations the Python way
Collection interfaces

Whenever you see method calls like the following, there is a quicker way to do it:

```python
enm = obj.createEnumeration()
while enm.hasMoreElements():
    value = enm.nextElement()
...
```

Instead, do:

```python
for value in obj:
    ...
```
Example: iterating over document paragraphs – the old way

doc = ... # Load a text document
enm = doc.Text.createEnumeration()
while enm.hasMoreElements():
    paragraph = enm.nextElement()
    print(paragraph.String)
Example: iterating over document paragraphs – the new way

```python
doc = ...  # Load a text document
for paragraph in doc.Text:
    print(paragraph.String)
```

Or use a Python style explicit iterator:

```python
doc = ...  # Load a text document
itr = iter(doc.Text)
paragraph = next(itr)
print(paragraph.String)
```

Or flatten the text so it can be accessed by index:

```python
doc = ...  # Load a text document
paragraphs = list(doc.Text)
print(paragraphs[0].String)
```

Obviously this can be inefficient for a large document – but extremely convenient in the context of e.g. a short test when there are only a few paragraphs
Elimination of explicit Any

- Certain method calls need to be passed an Any with a sequence of a specific type
- Most commonly this occurs with collection interfaces
- The syntax to deal with this in PyUNO was obscure and annoying

Example: creating a document index

```python
doc = ... # Load a text document
index = doc.createInstance("com.sun.star.text.ContentIndex");
uno.invoke(index.LevelParagraphStyles, \"replaceByIndex\", (0, uno.Any("[]string", ('Caption',))))
```

- PyUNO can now infer the type required by the collection automatically
  ```python
  index.LevelParagraphStyles[0] = ('Caption',)
  ```
List and iterator arguments

Wherever a UNO API expects a sequence, a Python list or iterator can now be passed.

This enables the use of list comprehensions and generator expressions for method calls and property assignments.

Example: Populate a text table

```python
doc = ... # Load a text document
tbl = doc.createInstance('com.sun.star.text.TextTable')
tbl.initialize(10,10)
doc.Text.insertTextContent(doc.CurrentController.ViewCursor, tbl, False)
# Assign numbers 0..99 to the cells using a generator expression
tbl.Data = ((y for y in range(10*x,10*x + 10)) for x in range(10))
```
List and iterator arguments

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
</tr>
<tr>
<td>40</td>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
<td>46</td>
<td>47</td>
<td>48</td>
<td>49</td>
</tr>
<tr>
<td>50</td>
<td>51</td>
<td>52</td>
<td>53</td>
<td>54</td>
<td>55</td>
<td>56</td>
<td>57</td>
<td>58</td>
<td>59</td>
</tr>
<tr>
<td>60</td>
<td>61</td>
<td>62</td>
<td>63</td>
<td>64</td>
<td>65</td>
<td>66</td>
<td>67</td>
<td>68</td>
<td>69</td>
</tr>
<tr>
<td>70</td>
<td>71</td>
<td>72</td>
<td>73</td>
<td>74</td>
<td>75</td>
<td>76</td>
<td>77</td>
<td>78</td>
<td>79</td>
</tr>
<tr>
<td>80</td>
<td>81</td>
<td>82</td>
<td>83</td>
<td>84</td>
<td>85</td>
<td>86</td>
<td>87</td>
<td>88</td>
<td>89</td>
</tr>
<tr>
<td>90</td>
<td>91</td>
<td>92</td>
<td>93</td>
<td>94</td>
<td>95</td>
<td>96</td>
<td>97</td>
<td>98</td>
<td>99</td>
</tr>
</tbody>
</table>
Tolerant struct initialisation

- Initialising a UNO struct previously required all members to be set, or none

- Example: PropertyValue – frequently, only name and value are needed

```python
from com.sun.star.beans import PropertyValue
prop1 = PropertyValue()
prop1.Name = 'foo'
prop1.Value = 'bar'
prop2 = PropertyValue('foo', 0, 'bar', 0)
prop3 = PropertyValue(Name='foo', Handle=0, Value='bar', State=0)
```

- This requirement is now relaxed when all arguments are named

```python
prop4 = PropertyValue(Name='foo', Value='bar')
```
Cell ranges

- A custom behaviour is applied to cell range objects
  - `com::sun::star::table::XCellRange`

- This is different to the other changes – the collection interfaces are generic, this is a higher level API
  - However, it's one that is widely used and could benefit from some syntactic sugar

- Applies to:
  - Sheets in Calc spreadsheets
  - Writer text tables
  - Subset cell ranges created on these
Cell ranges

**Existing syntax**

```python
cell = cellrange.getCellByPosition(col, row)
rng = cellrange.getCellRangeByPosition(left, top, right, bottom)
rng = cellrange.getCellRangeByName(name)
```

**New syntax – access like a two dimensional array**

```python
cell = cellrange[0,0]  # Access cell by indices
rng = cellrange[0,1:2]  # Access cell range by index,slice
rng = cellrange[1:2,0]  # Access cell range by slice,index
rng = cellrange[0:1,2:3]  # Access cell range by slices
rng = cellrange['A1:B2']  # Access cell range by descriptor
rng = cellrange['Name']  # Access cell range by name
```
Cell ranges

- Note that the indices used are in **Python/C order**
- These pairs are equivalent:

```python
# row r, column c
cell = cellrange[r,c]
cell = cellrange.getCellByPosition(c,r)

# rows t to b, columns l to r
rng = cellrange[t:b,l:r]
rng = cellrange.getCellRangeByPosition(l,t,r-1,b-1).
```
Cell ranges

- Objects which also implement com::sun::star::table::XColumnRowRange support negative indices (from-end indexing) and the below syntax for referencing whole rows and columns

- Calc spreadsheet sheets and cell ranges created upon these support this interface

- Writer text tables unfortunately don't

```python
rng = cellrange[0]          # Access cell range by row index
rng = cellrange[0,:]        # Access cell range by row index
rng = cellrange[:0]         # Access cell range by column index
```
Import constants by group name

Previously, UNO constants had to be imported individually

Example

```python
from com.sun.star.accessibility.AccessibleRole import MENU_BAR
from com.sun.star.accessibility.AccessibleRole import DIALOG
from com.sun.star.accessibility.AccessibleRole import PUSH_BUTTON
```

Constant groups can now be imported as a whole

```python
from com.sun.star.accessibility import AccessibleRole
# Now you can reference AccessibleRole.MENU_BAR etc.
```
Object hashability

- UNO objects should now have stable hash values
- This allows them to be safely used as keys for collections
  
  ```python
  s = set()
  s[obj] = 1
  ...
  # Later, we get the same object from UNO again
  # This only works if the object has a stable hash
  del s[obj]
  ```

- What's that “should” doing there?
  - Handle with care, don't rely on this if possible
  - Cases where this is useful should be rare
Performance improvements

- Every time a UNO object is passed to PyUNO, we have to perform introspection on it to find out information about its methods and properties.
- In the case of remote (out of process) PyUNO, this means making inter-process calls.
- Inter-process calls are slow, so the fewer the better.
- We can't avoid making at least a few calls.
- Up to LibreOffice 5.0 there was a bug which meant there were up to 50 inter-process calls for each object.
- Predictably this wasn't very fast.
- Further optimisations made to eliminate unnecessary calls and make some others lazy (only when actually needed, not for every object).
- Now it's much faster remotely and a little faster locally.
Fallout

- A major aim of these changes was not to break existing code
- Successful? Almost
  - Caused an issue with LibreLogo – commit 181a7b27acf29a2728be5a0eb3696796bc7df3da
  - Now that some PyUNO objects behave like proper Python collections, they have truth values that depend on whether or not they're empty
  - The LibreLogo code used a variable that was either 0 or a PyUNO object, and expected the two choices to be always False or True respectively
  - Mea culpa – didn't expect that
  - Unfortunately no easy way to work around
Questions?
Thank you