Situation

PyMuPDF is a Python binding for MuPDF, a library written in C. By nature of this construct, there obviously co-exist two levels of code: object-oriented Python code, which serves as an API to the programmer and the C-functions of MuPDF, which do the actual work.

The glue between these levels is generated by SWIG. SWIG translates MuPDF's C structures and functions into Python classes and their methods, respectively. For example, calling the C function `fz_open_document` not only opens a document for access, but also also creates a `Document` object in the Python code. A plethora of other functions dealing with opened documents in MuPDF are being made available as methods and properties of the created `Document` object at the same time. Examples of such functions are decryption, determining number of pages, accessing meta information and loading pages for further processing - another example of a Python object creation: a `Page` object.

While all this works marvelously as long as everybody is behaving well, it is at the same time frighteningly easy to provoke Python interpreter crashes by seemingly small errors.

An example

Obviously, a `1:n` relationship exists between a document and its pages. When a document is closed, most of its resources on the C-level are being freed. But in Python any *Page* objects are **not** deleted. Most or all `Page` methods and properties are disabled by the closed status of the parent `Document`, but the object as such continues to exist.

Premature Deletion of Document object

If the programmer deletes his `Document` object without closing it, an erroneous situation may result:

1. The Python document object is not accessible anymore.
2. Any *Page* objects have no knowledge about this and will show unexpected behaviour when being accessed.
3. In MuPDF's C code, the opened document continues to exist. There is no way to access it again during the current interpreter session. Any changes applied to it will be lost.

Complication

Methods `select()`, `deletePage()`, etc.

These methods change the structure of the current document and will very probably invalidate any existing `Page` object.

For similar reasons as in the previous section, accessing `Page` objects created before method execution afterwards will probably crash the interpreter.

There is even more ...

The annotations of a page (including links) also are in a `1:n` relationship to their page. When the owning page gets deleted, nothing takes care of these orphans currently.

Solution

We need a mechanism similar to **referential integrity** of relational databases.

But a simpler version suffices as follows:

1. Before a document is closed or deleted, all existing page objects should be deleted (or at least be marked as invalid for any access).

2. Likewise, before execution of any of the methods `select()`, `copyPage()`, `movePage()`, `deletePage()` and `deletePageRange()`, **all (!)** existing page objects should be deleted.
3. Similarly, before a page object is deleted, all of its annotations and links should be deleted. As page objects will be deleted via the previous two conditions, a cascading effect will result, when a document is closed.

Point 2. above deserves some comment to address objections like "why should page objects be deleted that are not affected by those methods?".

Short answer: In most cases, there are no unaffected page objects. Long answer:

1. Methods *select()* et al. work on the document level and by page number. They are not connected to any *Page* object. It would be very hard or impossible to find out whether there exist page objects whose numbers are among those deleted by these methods. Additionally, when pages get deleted, copied or moved, not only their own page number will change, but also those of many others.
2. Working *on the document level* means dealing with the document structure. It seems improbable, that `Page` level work like rendering or text extraction should occur at the same time or otherwise interfere. Because the above methods implicitely also change page numbers across the document, there is no reasonable way for the programmer to keep track of this.

So the easiest and clearest approach seems to be the suggested one.

Technical Approach

The Document and its Pages

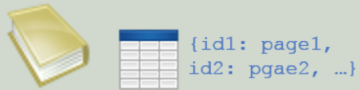
- When a `Page` is (`loadPage()`) created, it records itself in the document-owned dictionary `_page_refs = {id(page): page, ...}`. This dictionary is of type `weakref.WeakValueDictionary` to make sure pages can be deleted even if they are recorded here.
- Currently, page objects reference back to their containing document object (via `page.parent = document`). In order to intercept any unsolicited `del doc` or `doc = None` statement, this referencing is replaced by a weak reference (`weakref.proxy(doc)`).
- When all page objects get deleted:
    - Loop through the dictionary, and delete / invalidate each page object.
    - Empty the dictionary.
- When a `Page` is deleted, it deletes its entry in the `Document`'s dictionary via its object-`id`.

The Page and its Annotations / Links

This works much the same way as in before section:

- When an annotation or link is created, it records itself in its page's dictionary `_annot_refs = {id(annot): annot, ...}`. This dictionary is of type `weakref.WeakValueDictionary` too.
- Annotations and links reference their containing page via `annot.parent = page`. This is replaced by weak reference `annot.parent = weakref.proxy(page)`.
- When an `Annot` is deleted, it deletes its entry in the page's dictionary.

Document

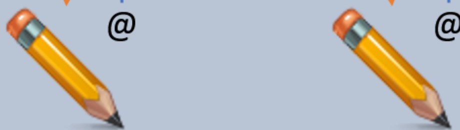


Dictionary _page_refs = {id(page): page, …} contains an entry for each page object in existence.
Dictionary is emptied by execution of methods close(), select(), deletePage(), deletePageRange(), copyPage(), movePage() and when document is deleted. In this event, method __del__() of each page entry is invoked.

Page

Dictionary _annot_refs = {id(annot): annot, …} contains an entry for each annot or link object in existence.
Dictionary is emptied when page object is deleted, i.e. its __del__() method is invoked. In this event, method __del__() of each entry is invoked.
Method __del__() of page objects also executes fz_drop_page if thisown == True. Sets parent = None and thisown = False. Deletes its entry in _page_refs.
Page objects' parent property points to their document via weakref.proxy.

Annotation



Method __del__() of each annotation or link object executes fz_drop_annot / fz_drop_link if thisown == True. Sets parent = None and thisown = False. Deletes its entry in _annot_refs.
Annotation and link objects' parent properties point to their page object via weakref.proxy.

Legend

Dictionary of type weakref.WeakValueDictionary

Pointer to parent of type weakref.proxy

Weak connection to parent object

Weak connection to child object

API changes

PyMuPDF users do not need to change anything and should even not realize that anything has changed - provided that no flaws in their script's flow exist.

However, if a page, annotation or link is referenced for which the respective parent has gone, a `RuntimeException` is now raised with text `orphaned object: parent is None`. This situation can be obviously be avoided by checking whether `parent` is `None`.

Because dependent objects are now being recorded and only weak references are being used, the Python and C levels of PyMuPDF applications should be kept much more in parallel. When a document is not referenced elsewhere, then whether `doc.close()`, `doc = None` or `del doc` are issued, all page objects and their respective annotations are dropped (freed) on the C level and become unusable on the Python level.