# Text Frames Interfere with Adjoining Borders

This document is provided to give a 2nd working copy of the same bug as reported in Python3docs Bugs #05, with extra information supplied.

## Comment

The report (below) was made at BugZilla#164302 on Dec 12 2024. The bug was first documented at GitHub Python3Docs Bug#05 and then documented separately as Bug#05.

a) Version 24.8.3.2 (Build) is affected by this bug
b) Grandmother version 3.3.0.4 is affected by this bug
c) All other versions between 3.3 & 24.8 are affected by this bug.
d) Michael Stahl declared that *"this is working as designed"*™ (good grief)

I now understand that the reason that LibreOffice has been riddled with bugs for the last 7+ years is *by design* rather than through ignorance, laziness or inability. There now follows extra comment + demonstration of this bug:–

*original bug report:*

In Writer, the bottom border of a Sidebar (a frame that contains text) will coalesce with the top inline-border of a paragraph. In other words, the below-frame spacing of the frame will be added to the internal top-spacing of the next paragraph, rather than being placed between the two entity's borders. This is almost certainly a dupe of 164297 and of 164298.

*Extra comment:*

The original report is almost accurate (if wordy), though it is probably more accurate to say (as in the H1 Heading at top) that a Text Frame *"Interferes"* with adjoining borders. It also helps if a person can see this in action. These are the features to pay attention to:–

1. I believe this may be a bug common to all framed entities.
2. The bug seems to occur with *anything* that is framed and is positioned next to something with a border.
3. The point below is that 3 of the paragraphs have a blue inline-border & thus the bug becomes discernable.
4. There are 5 demonstration pages below; the 1st two pages are individual examples of the two ingredients, whilst the final 3 pages are snapshots in a very short film.

## How to produce this document

1. This document created under 24.8.3.2.
   `menu:Help | About LibreOffice` shows: Build
2. The entire document bug04+.odt copied as this doc & then edited.
3. The SideBar "sidebar-06.01 The __del__() Special Method" on p18 within chapter_06.odt then copy/pasted into page#5 of this doc as a replacement for the former framed table + the outer frame changed from a thin blue to a #0A2746 dotted border (to contrast with the *CB2* paragraphs).

   There is 1 paragraph on that page. The text "some words" is placed below the SideBar simply to visually indicate where the Pilcrow ends.

   Paragraph pilcrows are displayed via either `Ctrl+F10` or by placing a tick into `menu:View | Formatting Marks`.
4. 4 paragraphs from p8 of chapter_01.odt were left as-is within page#6. (the *last 3 paragraphs are* Code Box 2 (*CB2*) *paragraphs, which have an inline text-border*)
5. The SideBar on page#5 was copy/pasted as a replacement for the former framed table in page#7.

   The end result is the words "para1", "para2" at the top of page#7, followed by the SideBar, then "para3", then a plain-text paragraph, and finally a *CB2* 3-line paragraph (with embedded blue border).
6. For the next 2 pages (page#8 + page#9) the whole of page#7 was copy/pasted as a replacement for the former contents. Then for each of those pages, the SideBar was selected on the page and then the down-arrow key (↓) was pressed (`<shift>+↓` & `<alt>+↓` also work).
7. For page#8 the key was pressed until the words "Let's look … the details:" appeared above the SideBar
8. For page#9 the key continued to be pressed until the *CB2* top-border appeared above the SideBar. That was easy to do, unlike with a framed table.

## *Demonstrations:*

Refer to the final pages (p#8 to p#9 ) for a practical demonstration of this bug.

*Things to note:–*

i. to recap:

   a) Each SideBar in all example pages is identical other than name.

   b) Further, each SideBar has Wrap is Optimal; spacing is 0.20cm left/right (internal) + bottom (external) = 0.5 cm.

   c) All text paragraphs are identical in borders + spacing to the SideBar, with the sole extra wrinkle that *Code Box 2* (*CB2*) paragraphs have the check-box selected for *“do not add space between paragraphs of the same style”* (which means that only the bottom paragraph has a bottom spacing).

   d) The combo of the above means that text elements can ’flow’ around the SideBar. In addition, it also means that, once selected, the SideBar can be moved up & down the ladder of the surrounding text paragraphs by means of the keyboard up- (↑) or down-arrow (↓); in this instance <shift> / <alt>+ arrow are also effective.

ii. in page#7 *(SideBar 5.2):*

   a) The bottom spacing below ALL paragraphs + image is 0.5 cm

   b) The 3 x *CB2* paragraphs are surrounded by an embedded blue border & the spacing from text above to the top of the border is 0.5 cm

iii. in page#8 *(SideBar 5.3):*

   a) This is the clearest demonstration of the bug.

   b) The text paragraph “Let’s look … the details:” is above the SideBar whilst the *CB2* paragraphs are immediately below.

   c) The spacing above the table looks a little too small whilst that below looks a little too large.

   d) Where has the *CB2* top-border gone? The frame lower-spacing is supposed to be between the bottom of the table & the top of the next paragraph. Instead, the latter is Missing In Action.

iv. in page#9 *(SideBar 5.4):*

a) This is probably the most blatant demonstration of the bug.

b) The *CB2* top-border now reappears above the image. I believe this to be a display error, in that the *CB2* border *'belongs'* to the *CB2* text & should not be separated from it. The current display looks daft.

c) This daft bug has been in place for more than 7 years, and is a Grandparent gift from Apache (if I recall the heredity of LibreOffice correctly). Time to fix it, methinks.

v. All frames show this bug.

## *Examples*

(see next pages):

## *The __del__() Special Method*

The `__del__(self)` special method is called when an object is destroyed—at least in theory. In practice, `__del__()` may never be called, even at program termination. Furthermore, when we write `del x`, all that happens is that the object reference `x` is deleted and the count of how many object references refer to the object that was referred to by `x` is decreased by `1`. Only when this count reaches `0` is `__del__()` likely to be called, but Python offers no guarantee that it will ever be called. In view of this, `__del__()` is very rarely reimplemented—none of the examples in this book reimplements it—and it should not be used to free up resources, so it is not suitable to be used for closing files, disconnecting network connections, or disconnecting database connections.

Python provides two separate mechanisms for ensuring that resources are properly released. One is to use a `try … finally` block as we have seen before and will see again in Chapter 7 [299 ➤], and the other is to use a context object in conjunction with a `with` statement—this is covered in Chapter 8 [347 ➤].

some words

Let’s look at a few tiny examples, and then discuss some of the details:

```
x = "blue"
y = "green"
z = x
```

para1

para2

### *The __del__() Special Method*

The `__del__(self)` special method is called when an object is destroyed—at least in theory. In practice, `__del__()` may never be called, even at program termination. Furthermore, when we write `del x`, all that happens is that the object reference `x` is deleted and the count of how many object references refer to the object that was referred to by `x` is decreased by `1`. Only when this count reaches `0` is `__del__()` likely to be called, but Python offers no guarantee that it will ever be called. In view of this, `__del__()` is very rarely reimplemented—none of the examples in this book reimplements it—and it should not be used to free up resources, so it is not suitable to be used for closing files, disconnecting network connections, or disconnecting database connections.

Python provides two separate mechanisms for ensuring that resources are properly released. One is to use a `try … finally` block as we have seen before and will see again in Chapter 7 [299 ➤], and the other is to use a context object in conjunction with a `with` statement—this is covered in Chapter 8 [347 ➤].

para3

Let’s look at a few tiny examples, and then discuss some of the details:

```
x = "blue"
y = "green"
z = x
```

para1

para2

para3

Let’s look at a few tiny examples, and then discuss some of the details:

## *The __del__() Special Method*

The `__del__(self)` special method is called when an object is destroyed—at least in theory. In practice, `__del__()` may never be called, even at program termination. Furthermore, when we write `del x`, all that happens is that the object reference `x` is deleted and the count of how many object references refer to the object that was referred to by `x` is decreased by `1`. Only when this count reaches `0` is `__del__()` likely to be called, but Python offers no guarantee that it will ever be called. In view of this, `__del__()` is very rarely reimplemented—none of the examples in this book reimplements it—and it should not be used to free up resources, so it is not suitable to be used for closing files, disconnecting network connections, or disconnecting database connections.

Python provides two separate mechanisms for ensuring that resources are properly released. One is to use a `try … finally` block as we have seen before and will see again in Chapter 7 [299 ➤], and the other is to use a context object in conjunction with a `with` statement—this is covered in Chapter 8 [347 ➤].

```
x = "blue"
y = "green"
z = x
```

para1

para2

para3

Let’s look at a few tiny examples, and then discuss some of the details:

## *The __del__() Special Method*

The `__del__(self)` special method is called when an object is destroyed—at least in theory. In practice, `__del__()` may never be called, even at program termination. Furthermore, when we write `del x`, all that happens is that the object reference `x` is deleted and the count of how many object references refer to the object that was referred to by `x` is decreased by `1`. Only when this count reaches `0` is `__del__()` likely to be called, but Python offers no guarantee that it will ever be called. In view of this, `__del__()` is very rarely reimplemented—none of the examples in this book reimplements it—and it should not be used to free up resources, so it is not suitable to be used for closing files, disconnecting network connections, or disconnecting database connections.

Python provides two separate mechanisms for ensuring that resources are properly released. One is to use a `try` … `finally` block as we have seen before and will see again in Chapter 7 [299 ➤], and the other is to use a context object in conjunction with a `with` statement—this is covered in Chapter 8 [347 ➤].

```
x = "blue"
y = "green"
z = x
```