Quickstart¶

Getting started with python-docx is easy. Let’s walk through the basics.

Opening a document

First thing you’ll need is a document to work on. The easiest way is this:

from docx import Document

document = Document()

This opens up a blank document based on the default “template”, pretty much what you get when you start a new document in Word using the built-in defaults. You can open and work on an existing Word document using python-docx, but we’ll keep things simple for the moment.

Adding a paragraph

Paragraphs are fundamental in Word. They’re used for body text, but also for headings and list items like bullets.

Here’s the simplest way to add one:

paragraph = document.add\_paragraph('Lorem ipsum dolor sit amet.')

This method returns a reference to a paragraph, newly added paragraph at the end of the document. The new paragraph reference is assigned to paragraph in this case, but I’ll be leaving that out in the following examples unless I have a need for it. In your code, often times you won’t be doing anything with the item after you’ve added it, so there’s not a lot of sense in keep a reference to it hanging around.

It’s also possible to use one paragraph as a “cursor” and insert a new paragraph directly above it:

prior\_paragraph = paragraph.insert\_paragraph\_before('Lorem ipsum')

This allows a paragraph to be inserted in the middle of a document, something that’s often important when modifying an existing document rather than generating one from scratch.

Adding a heading

In anything but the shortest document, body text is divided into sections, each of which starts with a heading. Here’s how to add one:

document.add\_heading('The REAL meaning of the universe')

By default, this adds a top-level heading, what appears in Word as ‘Heading 1’. When you want a heading for a sub-section, just specify the level you want as an integer between 1 and 9:

document.add\_heading('The role of dolphins', level=2)

If you specify a level of 0, a “Title” paragraph is added. This can be handy to start a relatively short document that doesn’t have a separate title page.

Adding a page break

Every once in a while you want the text that comes next to go on a separate page, even if the one you’re on isn’t full. A “hard” page break gets this done:

document.add\_page\_break()

If you find yourself using this very often, it’s probably a sign you could benefit by better understanding paragraph styles. One paragraph style property you can set is to break a page immediately before each paragraph having that style. So you might set your headings of a certain level to always start a new page. More on styles later. They turn out to be critically important for really getting the most out of Word.

Adding a table

One frequently encounters content that lends itself to tabular presentation, lined up in neat rows and columns. Word does a pretty good job at this. Here’s how to add a table:

table = document.add\_table(rows=2, cols=2)

Tables have several properties and methods you’ll need in order to populate them. Accessing individual cells is probably a good place to start. As a baseline, you can always access a cell by its row and column indicies:

cell = table.cell(0, 1)

This gives you the right-hand cell in the top row of the table we just created. Note that row and column indicies are zero-based, just like in list access.

Once you have a cell, you can put something in it:

cell.text = 'parrot, possibly dead'

Frequently it’s easier to access a row of cells at a time, for example when populating a table of variable length from a data source. The .rows property of a table provides access to individual rows, each of which has a .cells property. The .cells property on both Row and Column supports indexed access, like a list:

row = table.rows[1]

row.cells[0].text = 'Foo bar to you.'

row.cells[1].text = 'And a hearty foo bar to you too sir!'

The .rows and .columns collections on a table are iterable, so you can use them directly in a for loop. Same with the .cells sequences on a row or column:

for row in table.rows:

for cell in row.cells:

print(cell.text)

If you want a count of the rows or columns in the table, just use len() on the sequence:

row\_count = len(table.rows)

col\_count = len(table.columns)

You can also add rows to a table incrementally like so:

row = table.add\_row()

This can be very handy for the variable length table scenario we mentioned above:

# get table data -------------

items = (

(7, '1024', 'Plush kittens'),

(3, '2042', 'Furbees'),

(1, '1288', 'French Poodle Collars, Deluxe'),

)

# add table ------------------

table = document.add\_table(1, 3)

# populate header row --------

heading\_cells = table.rows[0].cells

heading\_cells[0].text = 'Qty'

heading\_cells[1].text = 'SKU'

heading\_cells[2].text = 'Description'

# add a data row for each item

for item in items:

cells = table.add\_row().cells

cells[0].text = str(item.qty)

cells[1].text = item.sku

cells[2].text = item.desc

The same works for columns, although I’ve yet to see a use case for it.

Word has a set of pre-formatted table styles you can pick from its table style gallery. You can apply one of those to the table like this:

table.style = 'LightShading-Accent1'

The style name is formed by removing all the spaces from the table style name. You can find the table style name by hovering your mouse over its thumbnail in Word’s table style gallery.

Adding a picture

Word lets you place an image in a document using the Insert > Photo > Picture from file... menu item. Here’s how to do it in python-docx:

document.add\_picture('image-filename.png')

This example uses a path, which loads the image file from the local filesystem. You can also use a file-like object, essentially any object that acts like an open file. This might be handy if you’re retrieving your image from a database or over a network and don’t want to get the filesystem involved.

Image size

By default, the added image appears at native size. This is often bigger than you want. Native size is calculated as pixels / dpi. So a 300x300 pixel image having 300 dpi resolution appears in a one inch square. The problem is most images don’t contain a dpi property and it defaults to 72 dpi. This would make the same image appear 4.167 inches on a side, somewhere around half the page.

To get the image the size you want, you can specify either its width or height in convenient units, like inches or centimeters:

from docx.shared import Inches

document.add\_picture('image-filename.png', width=Inches(1.0))

You’re free to specify both width and height, but usually you wouldn’t want to. If you specify only one, python-docx uses it to calculate the properly scaled value of the other. This way the aspect ratio is preserved and your picture doesn’t look stretched.

The Inches and Cm classes are provided to let you specify measurements in handy units. Internally, python-docx uses English Metric Units, 914400 to the inch. So if you forget and just put something like width=2 you’ll get an extremely small image :). You’ll need to import them from the docx.shared sub-package. You can use them in arithmetic just like they were an integer, which in fact they are. So an expression like width = Inches(3) / thing\_count works just fine.

Applying a paragraph style

If you don’t know what a Word paragraph style is you should definitely check it out. Basically it allows you to apply a whole set of formatting options to a paragraph at once. It’s a lot like CSS styles if you know what those are.

You can apply a paragraph style right when you create a paragraph:

document.add\_paragraph('Lorem ipsum dolor sit amet.', style='ListBullet')

This particular style causes the paragraph to appear as a bullet, a very handy thing. You can also apply a style afterward. These two lines are equivalent to the one above:

paragraph = document.add\_paragraph('Lorem ipsum dolor sit amet.')

paragraph.style = 'List Bullet'

The style is specified using its style name, ‘List Bullet’ in this example. Generally, the style name is exactly as it appears in the Word user interface (UI).

Applying bold and italic

In order to understand how bold and italic work, you need to understand a little about what goes on inside a paragraph. The short version is this:

A paragraph holds all the block-level formatting, like indentation, line height, tabs, and so forth.

Character-level formatting, such as bold and italic, are applied at the run level. All content within a paragraph must be within a run, but there can be more than one. So a paragraph with a bold word in the middle would need three runs, a normal one, a bold one containing the word, and another normal one for the text after.

When you add a paragraph by providing text to the .add\_paragraph() method, it gets put into a single run. You can add more using the .add\_run() method on the paragraph:

paragraph = document.add\_paragraph('Lorem ipsum ')

paragraph.add\_run('dolor sit amet.')

This produces a paragraph that looks just like one created from a single string. It’s not apparent where paragraph text is broken into runs unless you look at the XML. Note the trailing space at the end of the first string. You need to be explicit about where spaces appear at the beginning and end of a run. They’re not automatically inserted between runs. Expect to be caught by that one a few times :).

Run objects have both a .bold and .italic property that allows you to set their value for a run:

paragraph = document.add\_paragraph('Lorem ipsum ')

run = paragraph.add\_run('dolor')

run.bold = True

paragraph.add\_run(' sit amet.')

which produces text that looks like this: ‘Lorem ipsum dolor sit amet.’

Note that you can set bold or italic right on the result of .add\_run() if you don’t need it for anything else:

paragraph.add\_run('dolor').bold = True

# is equivalent to:

run = paragraph.add\_run('dolor')

run.bold = True

# except you don't have a reference to `run` afterward

It’s not necessary to provide text to the .add\_paragraph() method. This can make your code simpler if you’re building the paragraph up from runs anyway:

paragraph = document.add\_paragraph()

paragraph.add\_run('Lorem ipsum ')

paragraph.add\_run('dolor').bold = True

paragraph.add\_run(' sit amet.')

Applying a character style

In addition to paragraph styles, which specify a group of paragraph-level settings, Word has character styles which specify a group of run-level settings. In general you can think of a character style as specifying a font, including its typeface, size, color, bold, italic, etc.

Like paragraph styles, a character style must already be defined in the document you open with the Document() call (see Understanding Styles).

A character style can be specified when adding a new run:

paragraph = document.add\_paragraph('Normal text, ')

paragraph.add\_run('text with emphasis.', 'Emphasis')

You can also apply a style to a run after it is created. This code produces the same result as the lines above:

paragraph = document.add\_paragraph('Normal text, ')

run = paragraph.add\_run('text with emphasis.')

run.style = 'Emphasis'

As with a paragraph style, the style name is as it appears in the Word UI.

Working with Documents

python-docx allows you to create new documents as well as make changes to existing ones. Actually, it only lets you make changes to existing documents; it’s just that if you start with a document that doesn’t have any content, it might feel at first like you’re creating one from scratch.

This characteristic is a powerful one. A lot of how a document looks is determined by the parts that are left when you delete all the content. Things like styles and page headers and footers are contained separately from the main content, allowing you to place a good deal of customization in your starting document that then appears in the document you produce.

Let’s walk through the steps to create a document one example at a time, starting with two of the main things you can do with a document, open it and save it.

Opening a document

The simplest way to get started is to open a new document without specifying a file to open:

from docx import Document

document = Document()

document.save('test.docx')

This creates a new document from the built-in default template and saves it unchanged to a file named ‘test.docx’. The so-called “default template” is actually just a Word file having no content, stored with the installed python-docx package. It’s roughly the same as you get by picking the Word Document template after selecting Word’s File > New from Template… menu item.

REALLY opening a document

If you want more control over the final document, or if you want to change an existing document, you need to open one with a filename:

document = Document('existing-document-file.docx')

document.save('new-file-name.docx')

Things to note:

You can open any Word 2007 or later file this way (.doc files from Word 2003 and earlier won’t work). While you might not be able to manipulate all the contents yet, whatever is already in there will load and save just fine. The feature set is still being built out, so you can’t add or change things like headers or footnotes yet, but if the document has them python-docx is polite enough to leave them alone and smart enough to save them without actually understanding what they are.

If you use the same filename to open and save the file, python-docx will obediently overwrite the original file without a peep. You’ll want to make sure that’s what you intend.

Opening a ‘file-like’ document

python-docx can open a document from a so-called file-like object. It can also save to a file-like object. This can be handy when you want to get the source or target document over a network connection or from a database and don’t want to (or aren’t allowed to) interact with the file system. In practice this means you can pass an open file or StringIO/BytesIO stream object to open or save a document like so:

f = open('foobar.docx', 'rb')

document = Document(f)

f.close()

# or

with open('foobar.docx', 'rb') as f:

source\_stream = StringIO(f.read())

document = Document(source\_stream)

source\_stream.close()

...

target\_stream = StringIO()

document.save(target\_stream)

The 'rb' file open mode parameter isn’t required on all operating systems. It defaults to 'r' which is enough sometimes, but the ‘b’ (selecting binary mode) is required on Windows and at least some versions of Linux to allow Zipfile to open the file.

Okay, so you’ve got a document open and are pretty sure you can save it somewhere later. Next step is to get some content in there …