pdfme

Andres Felipe Sierra Parra

May 04, 2022

CONTENTS

This is a powerful library to create PDF documents easily.

The way you create a PDF document with pdfme is very similar to how you create documents with LaTex: you just tell pdfme at a very high level what elements you want to be in the document, without worrying about wrapping text in a box, positioning every element inside the page, creating the lines of a table, or the internals of the PDF Document Format. pdfme will put every element below the last one, and when a page is full it will add a new page to keep adding elements to the document, and will keep adding pages until all of the elements are inside the document. It just works.

If you want the power to place elements wherever you want and mess with the PDF Document Format internals, pdfme got you covered too. Give the docs a look to check how you can do this.

CHAPTER

ONE

MAIN FEATURES

- You can create a document without having to worry about the position of each element in the document. But you have the possibility to place any element wherever you want too.
- You can add rich text paragraphs (paragraphs with text in different sizes, fonts, colors and styles).
- You can add images.
- You can add tables and place whatever you want on their cells, span columns and rows, and change the fills and borders in the easiest way possible.
- You can add group elements that contain paragraphs, images or tables, and guarantee that all of the children elements in the group element will be in the same page.
- You can add content boxes, a multi-column element where you can add paragraphs, images, tables and even content boxes themselves. The elements inside this content boxes are added from top to bottom and from left to right.
- You can add url links (to web pages), labels/references, footnotes and outlines anywhere in the document.
- You can add running sections, content boxes that will be included in every page you add to the document. Headers and footers are the most common running sections, but you can add running sections anywhere in the page.

CHAPTER

TWO

INSTALLATION

You can install using pip:

```
pip install pdfme
```

CHAPTER

THREE

ABOUT THIS DOCS

We recommend starting with the tutorial in *Tutorial*, but you can find the description and instructions for each feature inside the docs for each class representing the feature, so in *pdfme.text.PDFText* class you'll learn how to build a paragraph, in *pdfme.table.PDFTable* class you'll learn how to build a table, in *pdfme.content.PDFContent* class you'll learn how to build a content box, in *pdfme.document.PDFDocument* class you'll learn how to build a PDF from a nested-dict structure (Json) and in *pdfme.pdf.PDF* class you'll learn how to use the main class of this library, the one that represents the PDF document.

CHAPTER

FOUR

USAGE

You can use this library to create PDF documents by using one of the following strategies:

- The recommended way is to use the function *pdfme.document.build_pdf()*, passing a dictionary with the description and styling of the document as its argument. *Tutorial* section uses this method to build a PDF document, and you can get more information about this approach in *pdfme.document.PDFDocument* definition.
- Use the *pdfme.pdf.PDF* class and use its methods to build the PDF document. For more information about this approach see *pdfme.pdf.PDF* class definition.

CHAPTER

FIVE

SHORTCOMINGS

- Currently this library only supports the standard 14 PDF fonts.
- Currently this library only supports `jpg` and `png` image formats (png images are converted to jpg images using Pillow, so you have to install it to be able to embed png images).

You can explore the rest of this library components in the following links:

5.1 Tutorial

In this tutorial we will create a PDF document using pdfme to showcase some of its functionalities.

We will use the preferred way to build a document in pdfme, that is using *pdfme.document.build_pdf()* function. This function receives as its first argument a nested dict structure with the contents and styling of the document, and most of this tutorial will be focused on building this dict.

In every step we will tell you the class or function definition where you can get more information.

Let's start importing the library and creating the root dictionary where the definitions that affect the whole document will be stated: `document`.

```
from pdfme import build_pdf

document = {}
```

Now add the `style` key to this dictionary, with the styling that all of the sections will inherit.

```
document['style'] = {
    'margin_bottom': 15,
    'text_align': 'j'
}
```

In our example we define the `margin_bottom` property, that will be the default space below every element in the document, and `text_align` will be the default text alignment for all the paragraphs in the document. In this dict you can set the default value for the style properties that affect the paragraphs (`text_align`, `line_height`, `indent`, `list_text`, `list_style`, `list_indent`, `b`, `i`, `s`, `f`, `u`, `c`, `bg`, `r`), images (`image_place`), tables (`cell_margin`, `cell_margin_left`, `cell_margin_top`, `cell_margin_right`, `cell_margin_bottom`, `cell_fill`, `border_width`, `border_color`, `border_style`) and content boxes (`margin_top`, `margin_left`, `margin_bottom`, `margin_right`) inside the document. For information about paragraph properties see *pdfme.text.PDFText*, about table properties see *pdfme.table.PDFTable*, and about image and content properties see *pdfme.content.PDFContent*.

You can set page related properties in `style` too, like `page_size`, `rotate_page`, `margin`, `page_numbering_offset` and `page_numbering_style` (see *pdfme.pdf.PDF* definition).

You can also define named style instructions or formats (something like CSS classes) in the `document` dict like this:

```
document['formats'] = {
    'url': {'c': 'blue', 'u': 1},
    'title': {'b': 1, 's': 13}
}
```

Every key in `formats` dict will be the name of a format that you will be able to use anywhere in the document. In the example above we define a format for urls, the typical blue underlined style, and a format for titles with a bigger font size and bolded text. Given you can use this formats anywhere, the properties you can add to them are the same you can add to the document's `style` we described before.

One more key you can add to `document` dict is `running_sections`. In here you can define named content boxes that when referenced in a section, will be added to every page of it. Let's see how we can define a header and footer for our document using running sections:

```
document['running_sections'] = {
    'header': {
        'x': 'left', 'y': 20, 'height': 'top',
        'style': {'text_align': 'r'},
        'content': [{'.b': 'This is a header'}]
    },
    'footer': {
        'x': 'left', 'y': 800, 'height': 'bottom',
        'style': {'text_align': 'c'},
        'content': [{'.': ['Page ', {'var': '$page'}]}]
    }
}
```

Here we defined running sections `header` and `footer`, with their respective positions and styles. To know more about running sections see *pdfme.document.PDFDocument* definition. We will talk about text formatting later, but one important thing to note here is the use of `$page` variable inside footer's `content`. This is the way you can include the number of the page inside a paragraph in pdfme.

Just defining these running sections won't add them to every page of the document; you will have to reference them in the section you want to really use them, or add a `per_page` dictionary like this:

```
document['per_page'] = [
    {'pages': '1:1000:2', 'style': {'margin': [60, 100, 60, 60]}},
    {'pages': '0:1000:2', 'style': {'margin': [60, 60, 60, 100]}},
    {'pages': '0:4:2', 'running_sections': {'include': ['header']}},
]
```

This dictionary will style, include or exclude running sections from the pages you set in the property `pages`. This key is a string of comma separated ranges of pages, and in this particular case we will add `header` to pages 0 and 2, and will add more left margin in odd pages, and more right margin in even pages. To know more about `per_page` dict see *pdfme.document.PDFDocument*. Keep reading to see how we add `header` and `footer` per sections.

Finally we are going to talk about *sections*. These can have their own page layout, page numbering, running sections and style, and are the places where we define the contents of the document. It's important to note that after every section there's a page break.

Let's create `sections` list to contain the documents sections, and add our first section `section1`.

```
document['sections'] = []
section1 = {}
```

(continues on next page)

(continued from previous page)

```
document['sections'].append(section1)
```

A section is just a content box, a multi-column element where you can add paragraphs, images, tables and even content boxes themselves (see *pdfme.content.PDFContent* for more informarion about content boxes). pdfme will put every element from a section in the PDF document from top to bottom, and when the first page is full it will add a new page to keep adding elements to the document, and will keep adding pages until all of the elements are inside the document.

Like a regular content box you can add a `style` key to a section, where you can reference a format (from the `formats` dict we created before), or add a new `style` dict, and with this you can overwrite any of the default style properties of the document.

```
section1['style'] = {
    'page_numbering_style': 'roman'
}
```

Here we overwrite only `page_numbering_style`, a property that sets the style of the page numbers inside the section (see *pdfme.pdf.PDF* definition). Default value is `arabic` style, and here we change it to `roman` (at least for this section).

Now we are going to reference the running sections that we will use in this section.

```
section1['running_sections'] = ['footer']
```

In this first section we will only use the `footer`. pdfme will add all of the running_sections referenced in `running_sections` list, in the order they are in this list, to every page of this section.

And finally we will define the contents of this section, inside `content1` list.

```
section1['content'] = content1 = []
```

We will first add a title for this section:

```
content1.append({
    '.': 'A Title', 'style': 'title', 'label': 'title1',
    'outline': {'level': 1, 'text': 'A different title 1'}
})
```

We added a paragraph dict, and it's itself what we call a paragraph part. A paragraph part can have other nested paragraph parts, as it's explained in *pdfme.text.PDFText* definition. This is like an HTML structure, where you can define a style in a root element and its style will be passed to all of its descendants.

The first key in this dictionary we added is what we call a dot key, and is where we place the contents of a paragraph part, and its descendants. We won't extend much on the format for paragraphs, as it's explained in *pdfme.text.PDFText* definition, so let's talk about the other keys in this dict. First we have a `style` key, with the name of a format that we defined before in the document's `formats` dict. This will apply all of the properties of that format into this paragraph part. We have a `label` key too, defining a position in the PDF document called `title1`. Thanks to this we will be able to navigate to this position from any place in the document, just by using a reference to this label (keep reading to see how we reference this title in the second section). Finally, we have an `outline` key with a dictionary defining a PDF outline, a position in the PDF document, to which we can navigate to from the outline panel of the pdf reader. More information about outlines in *pdfme.text.PDFText*.

Now we will add our first paragraph.

```
content1.append(
    ['This is a paragraph with a ', {'.b;c:green': 'bold green part'}, ', a ',
```

(continues on next page)

(continued from previous page)

```
    {'.': 'link', 'style': 'url', 'uri': 'https://some.url.com'},
    ', a footnote', {'footnote': 'description of the footnote'},
    ' and a reference to ',
    {'.': 'Title 2.', 'style': 'url', 'ref': 'title2'}]
)
```

Note that this paragraph is not a dict, like the title we added before. Here we use a list of paragraph parts, a shortcut when you have a paragraph with different styles or with labels, references, urls, outlines or footnotes.

We give format to the second paragraph part by using its dot key. This way of giving format to a paragraph part is something like the inline styles in HTML elements, and in particular in this example we are making the text inside this part bold and green.

The rest of this list paragraph parts are examples of how to add a url, a footnote and a reference (clickable links to go to the location in the document of the label we reference) to the second title of this document (located in the second section).

Next we will add an image to the document, located in the relative path `path/to/some_image.jpg`.

```
content1.append({
    'image': 'path/to/some_image.jpg',
    'style': {'margin_left': 100, 'margin_right': 100}
})
```

In `style` dict we set `margin_left` and `margin_right` to 100 to make our image narrower and center it in the page.

Next we will add a group element, containing an image and a paragraph with the image description. This guarantees that both the image and its description will be placed in the same page. To know more about group elements, and how to control the its height check *pdfme.content.PDFContent*.

```
content1.append({
    "style": {"margin_left": 80, "margin_right": 80},
    "group": [
        {"image": 'path/to/some_image.jpg'},
        {".": "Figure 1: Description of figure 1"}
    ]
})
```

Next we will add our first table to the document, a table with summary statistics from a database table.

```
table_def1 = {
    'widths': [1.5, 1, 1, 1],
    'style': {'border_width': 0, 'margin_left': 70, 'margin_right': 70},
    'fills': [{'pos': '1::2;:', 'color': 0.7}],
    'borders': [{'pos': 'h0,1,-1;:', 'width': 0.5}],
    'table': [
        ['', 'column 1', 'column 2', 'column 3'],
        ['count', '2000', '2000', '2000'],
        ['mean', '28.58', '2643.66', '539.41'],
        ['std', '12.58', '2179.94', '421.49'],
        ['min', '1.00', '2.00', '1.00'],
        ['25%', '18.00', '1462.00', '297.00'],
        ['50%', '29.00', '2127.00', '434.00'],
        ['75%', '37.00', '3151.25', '648.25'],
        ['max', '52.00', '37937.00', '6445.00']
```

(continues on next page)

(continued from previous page)

```
    ]
}

content1.append(table_def1)
```

In `widths` list we defined the width for every column in the table. The numbers here are not percentages or fractions but proportions. For example, in our table the first column is 1.5 times larger than the second one, and the third and fourth one are the same length as the second one.

In `style` dict we set the `border_width` of the table to 0, thus hiding all of this table lines. We also set `margin_left` and `margin_right` to 70 to make our table narrower and center it in the page.

In `fills` we overwrite the default value of `cell_fill`, for some of the rows in the table. The format of this `fills` list is explained in *pdfme.table.PDFTable* definition, but in short, we are setting the fill color of the even rows to a gray color.

In `borders` we overwrite the default value of `border_width` (which we set to 0 in `style`) for some of the horizontal borders in the table. The format of this `borders` list is explained in *pdfme.table.PDFTable* definition too, but in short, we are setting the border width of the first, second and last horizontal borders to 0.5.

And finally we are adding the table contents in the `table` key. Each list, in this `table` list, represents a row of the table, and each element in a row list represents a cell.

Next we will add our second table to the document, a form table with some cells combined.

```
table_def2 = {
    'widths': [1.2, .8, 1, 1],
    'table': [
        [
            {
                'colspan': 4,
                'style': {
                    'cell_fill': [0.8, 0.53, 0.3],
                    'text_align': 'c'
                },
                '.b;c:1;s:12': 'Fake Form'
            },None, None, None
        ],
        [
            {'colspan': 2, '.': [{'.b': 'First Name\n'}, 'Fakechael']}, None,
            {'colspan': 2, '.': [{'.b': 'Last Name\n'}, 'Fakinson Faker']}, None
        ],
        [
            [{'.b': 'Email\n'}, 'fakeuser@fakemail.com'],
            [{'.b': 'Age\n'}, '35'],
            [{'.b': 'City of Residence\n'}, 'Fake City'],
            [{'.b': 'Cell Number\n'}, '33333333333'],
        ]
    ]
}

content1.append(table_def2)
```

In the first row we combined the 4 columns to show the title of the form; in the second row we combine the first 2 columns for the first name, and the other 2 columns for the last name; and in the last row we use the four cells to the

rest of the information.

Notice that cells that are below or to the right of a merged cell must be equal to `None`, and that instead of using strings inside the cells, like we did in the first table, we used paragraph parts in the cells. And besides paragraphs you can add a content box, an image or even another table to a cell.

Now we will add a second section.

```
document['sections'].append({
    'style': {
        'page_numbering_reset': True, 'page_numbering_style': 'arabic'
    },
    'running_sections': ['header', 'footer'],
    'content': [

        {
            '.': 'Title 2', 'style': 'title', 'label': 'title2',
            'outline': {}
        },

        {
            'style': {'list_text': '1.  '},
            '.': ['This is a list paragraph with a reference to ',
            {'.': 'Title 1.', 'style': 'url', 'ref': 'title1'}]
        }
    ]
})
```

In this section we set the page numbering style back to the default value, `arabic`, and we reset the page count to 1 by including `page_numbering_reset` in the `style` dict.

We also added running section `header`, additional to the running section `footer` we used in the first section.

And we added the second title of the document, with its label and outline, and a list paragraph (a paragraph with text `'1. '` on the left of the paragraph) with a reference to the first title of the document.

Finally, we will generate the PDF document from the dict `document` we just built, by using `build_pdf` function.

```
with open('document.pdf', 'wb') as f:
    build_pdf(document, f)
```

Following these steps we will have a PDF document called `document.pdf` with all of the contents we added to `document` dict.

5.2 Examples

Example of a PDF document created with *pdfme.document.build_pdf()* using almost all of the functionalities of this library.

```
import random

from pdfme import build_pdf


random.seed(1)
```

(continues on next page)

(continued from previous page)

```
abc = 'abcdefghijklmnñopqrstuvwxyzABCDEFGHIJKLMNÑOPQRSTUVWXYZáéíóúÁÉÍÓÚ'

def gen_word():
    return ''.join(random.choice(abc) for _ in range(random.randint(1, 10)))

def gen_text(n):
    return random.choice(['',' ']) + (' '.join(gen_word() for _ in range(n))) + random.
↪choice(['',' '])

def gen_paragraphs(n):
    return [gen_text(random.randint(50, 200)) for _ in range(n)]

document = {
    "style": {
        "margin_bottom": 15, "text_align": "j",
        "page_size": "letter", "margin": [60, 50]
    },
    "formats": {
        "url": {"c": "blue", "u": 1},
        "title": {"b": 1, "s": 13}
    },
    "running_sections": {
        "header": {
            "x": "left", "y": 20, "height": "top", "style": {"text_align": "r"},
            "content": [{".b": "This is a header"}]
        },
        "footer": {
            "x": "left", "y": 740, "height": "bottom", "style": {"text_align": "c"},
            "content": [{".": ["Page ", {"var": "$page"}]}]
        }
    },
    "sections": [
        {
            "style": {"page_numbering_style": "roman"},
            "running_sections": ["footer"],
            "content": [

                {
                    ".": "A Title", "style": "title", "label": "title1",
                    "outline": {"level": 1, "text": "A different title 1"}
                },

                ["This is a paragraph with a ", {".b": "bold part"}, ", a ",
                {".": "link", "style": "url", "uri": "https://some.url.com"},
                ", a footnote", {"footnote": "description of the footnote"},
                " and a reference to ",
                {".": "Title 2.", "style": "url", "ref": "title2"}],

                {"image": "path/to/some/image.jpg"},

                *gen_paragraphs(7),
```

(continues on next page)

(continued from previous page)

```
                {
                    "widths": [1.5, 2.5, 1, 1.5, 1, 1],
                    "style": {"s": 9},
                    "table": [
                        [
                            gen_text(4),
                            {
                                "colspan": 5,
                                "style": {
                                    "cell_fill": [0.57, 0.8, 0.3],
                                    "text_align": "c", "cell_margin_top": 13
                                },
                                ".b;c:1;s:12": gen_text(4)
                            },None, None, None, None
                        ],
                        [
                            {"colspan": 2, ".": [{".b": gen_text(3)}, gen_text(3)]},␣
↪None,
                            {".": [{".b": gen_text(1) + "\n"}, gen_text(3)]},
                            {".": [{".b": gen_text(1) + "\n"}, gen_text(3)]},
                            {".": [{".b": gen_text(1) + "\n"}, gen_text(3)]},
                            {".": [{".b": gen_text(1) + "\n"}, gen_text(3)]}
                        ],
                        [
                            {
                                "colspan": 6, "cols": {"count": 3, "gap": 20},
                                "style": {"s": 8},
                                "content": gen_paragraphs(10)
                            },
                            None, None, None, None, None
                        ]
                    ]
                },

                *gen_paragraphs(10),
            ]
        },
        {
            "style": {
                "page_numbering_reset": True, "page_numbering_style": "arabic"
            },
            "running_sections": ["header", "footer"],
            "content": [

                {
                    ".": "Title 2", "style": "title", "label": "title2",
                    "outline": {}
                },

                ["This is a paragraph with a reference to ",
                {".": "Title 1.", "style": "url", "ref": "title1"}],
```

(continues on next page)

(continued from previous page)

```
                {
                    "style": {"list_text": "1.  "},
                    ".": "And this is a list paragraph." + gen_text(40)
                },

                *gen_paragraphs(10)
            ]
        },
    ]
}

with open('document.pdf', 'wb') as f:
    build_pdf(document, f)
```

5.3 Modules

5.3.1 pdfme.base

class pdfme.base.**PDFBase**(*version='1.5'*, *trailer=None*)

Bases: `object`

This class represents a PDF file, and deals with parsing python objects you add to it (with method `add`) to PDF indirect objects. The python types that are parsable to their equivalent PDF types are `dict` (parsed to PDF Dictionaries), `list`, `tuple`, `set` (parsed to PDF Arrays), `bytes` (no parsing is done with this type), `bool` (parsed to PDF Boolean), `int` (parsed to PDF Integer), `float` (parsed to PDF Real), `str` (parsed to PDF String) and `PDFObject`, a python representation of a PDF object.

When you are done adding objects to an instance of this class, you just have to call its `output` method to create the PDF file, and we will take care of creating the head, the objects, the streams, the xref table, the trailer, etc.

As mentioned before, you can use python type `bytes` to add anything to the PDF file, and this can be used to add PDF objects like *Names*.

For `dict` objects, the keys must be of type `str` and you don't have to use PDF Names for the keys, because they are automatically transformed into PDF Names when the PDF file is being created. For example, to add a page dict, the keys would be `Type`, `Content` and `Resources`, instead of `/Type`, `/Content` and `/Resources`, like this:

```
base = PDFBase()
page_dict = {
    'Type': b'/Page', 'Contents': stream_obj_ref, 'Resources': {}
}
base.add(page_dict)
```

You can add a `stream` object by adding a `dict` like the one described in function *pdfme.parser.parse_stream()*.

This class behaves like a `list`, and you can get a `PDFObject` by index (you can get the index from a `PDFObject.id` attribute), update by index, iterate through the PDF PDFObjects and use `len` to get the amount of objects in this list-like class.

Parameters

- **version** (*str, optional*) – Version of the PDF file. Defaults to '1.5'.

- **trailer** (*dict, optional*) – You can create your own trailer dict and pass it as this argument.

Raises **ValueError** – If trailer is not dict type

add(*py_obj*)

Add a new object to the PDF file

Parameters **py_obj** (*dict, list, tuple, set, bytes, bool, int, float, str,* PDFObject) – Object to be added.

Raises **TypeError** – If `py_obj` arg is not an allowed type.

Returns A PDFObject representing the object added

Return type *PDFObject*

output(*buffer*)

Create the PDF file.

Parameters **buffer** (*file_like*) – A file-like object to write the PDF file into.

5.3.2 pdfme.color

class pdfme.color.**PDFColor**(*color*, *stroke=False*)

Bases: `object`

Class that generates a PDF color string (with function `str()`) using the rules described in *pdfme.color.parse_color()*.

Parameters

- **color** (*int, float, list, tuple, str,* PDFColor) – The color specification.
- **stroke** (*bool, optional*) – Whether this is a color for stroke(True) or for fill(False). Defaults to False.

pdfme.color.**parse_color**(*color*)

Function to parse `color` into a list representing a PDF color.

The scale of the colors is between 0 and 1, instead of 0 and 256, so all the numbers in `color` must be between 0 and 1.

`color` of type int or float represents a gray color between black (0) and white (1).

`color` of type list or tuple is a gray color if its length is 1, a rgb color if its length is 3, and a rgba color if its length is 4 (not yet supported).

`color` of type str can be a hex color of the form "#aabbcc", the name of a color in the variable `colors` in file color.py, or a space separated list of numbers, that is parsed as an rgb color, like the one described before in the list `color` type.

Parameters **color** (*int, float, list, tuple, str*) – The color specification.

Returns list representing the PDF color.

Return type list

5.3.3 pdfme.content

class pdfme.content.**PDFContent**(*content*, *fonts*, *x*, *y*, *width*, *height*, *pdf=None*)

Bases: `object`

This class represents a group of elements (paragraphs, images, tables) to be added to a *pdfme.pdf.PDF* instance; what is called a "content box" in this library.

This class receives as the first argument a dict representing the layout of the elements that are going to be added to the PDF. This dict must have a `content` key with a tuple or a list as its value, containing the elements to be added.

The elements are arranged by using method `run` from top to bottom, and from left to right in order, in the rectangle defined by args `x`, `y`, `width` and `height`. The elements are added to this rectangle, until they are all inside of it, or until all of the vertical space is used and the rest of the elements can not be added. In these two cases method `run` finishes, and the property `finished` will be True if all the elements were added, and False if the vertical space ran out. If `finished` is False, you can set a new rectangle (on a new page for example) and use method `run` again (passing the parameters of the new rectangle) to add the remaining elements that couldn't be added in the last rectangle. You can keep doing this until all of the elements are added and therefore property `finished` is True.

By using method `run` the elements are not really added to the PDF object. After calling `run`, the properties `fills` and `lines` will be populated with the fills and lines of the tables that fitted inside the rectangle, and `parts` will be filled with the paragraphs and images that fitted inside the rectangle too, and you have to add them by yourself to the PDF object before using method `run` again (in case `finished` is False), because they will be redefined for the next rectangle after calling it again. You can check the `content` method in PDF module to see how this process is done.

This process of creating new pages to fit all of the elements of a content box is done automatically when you use *pdfme.document.PDFDocument* or :*pdfme.document.build_pdf()*

A `cols` key with a dict as a value can be included to arrange the elements in more than one column. For example, to use 2 columns, and to set a gap space between the 2 columns of 20 points, a dict like this one can be used:

```
{
    'cols': {'count': 2, 'gap': 20},
    'content': ['This is a lot of text ...'],
}
```

The elements in the `content` list can be one of the following:

- A paragraph that can be a string, a list, a tuple or a dictionary with a key starting with a `.`. To know more about paragraphs check *pdfme.text.PDFText*. Additional to the keys that can be included inside `style` key of a paragraph dict like the one you pass to *pdfme.text.PDFText*, you can include the following too: `text_align`, `line_height`, `indent`, `list_text`, `list_style` and `list_indent`. For information about these attributes check *pdfme.text.PDFText*. Here is an example of a paragraph dict:

  ```
  {
      'style': {
          'text_align': 'j',
          'line_height': 1.5,
          'list_text': '1. ',
      },
      '.b': 'This is a bold text.'
  }
  ```

  This paragraph dict yields a justified paragraph with a line height of 1.5 times the original line height and with a **1** on the left of the paragraph.

- An image that should be a dict with a `image` key, holding the path of the image, or the bytes of the image. In case `image` is of type bytes two more keys should be added to this dict: `image_name` and `extension`, being the first a unique name for the image and the second the extension or format of the image (ex. "jpg"). This dict can have a `style` dict, to tell this class what should it do when an image don't fit a column through the key `image_place`. This attribute can be "normal" or "flow"(default) and both of them will take the image to the next column or rectangle, but the second one will try to accommodate the elements coming after the image to fill the space left by it. Here is an example of an image dict:

```
{
    'style': { 'image_place': 'flow' },
    'image': '/path/to/an/image.jpg'
}
```

- A table that should be a dict with a `table` key with the table data, and optionally any or all of the following keys: `widths`, `borders` and `fills`. To know more about these keys check their meaning in *pdfme.table.PDFTable*. Here is an example of a table dict:

```
{
    'table': [['col1', 'col2', 'col3'], ['value1', 'value2', 'value3']],
    'widths': [1,2,3],
    'borders': [{'pos': 'h0,1,3;:', 'width': 2, 'color': 0}]
}
```

- A content box that can be a dict like the one being explained here, and can contain other elements inside it recursively. This can be used to insert a new section with more columns (for example a 2 columns content box, inside another 2 columns content box).

- A group element that is a list of paragraphs, images or tables that should be placed all in the same page. This can be used for example to place an image with a description, with the guarantee that both will be in the same page. Be careful though, because the group element should fit the width and max height of the containing box, or else an error will be raised. This can be "relaxed" by setting `min_height` property style in the images inside the group. If an image does not have this property it will take as much space as possible, and if it does it will be shrinked as much as possible (without shrinking it beyond `min_height`) to make the other elements in the group fit in the available height. If there are more than one images in the group with `min_height` style property they will be shrinked together proportionally. If you want to ensure that some image will be shrinked until its `min_height`, use the `shrink` style property. Here is an example of a group element:

```
{
    "style": {"margin_left": 80, "margin_right": 80},
    "group": [
        {"image": "tests/image_test.jpg", "min_height": 200},
        {".": "Figure 1: Description of figure 1"}
    ]
}
```

Each element in the content box can have margins to keep it separated from the other elements, and these margins can be set inside the `style` dict of the content box dict with the following keys: `margin_top`, `margin_left`, `margin_bottom` and `margin_right`. Default value for all of them is 0, except for `margin_bottom` that have a default value of 5.

All of the children elements in the content box will inherit the the content box style.

Parameters

- **content** (*dict*) – A content dict.

- **fonts** (PDFFonts) – A PDFFonts object used to build paragraphs.
- **x** (*int, float*) – The x position of the left of the rectangle.
- **y** (*int, float*) – The y position of the top of the rectangle.
- **width** (*int, float*) – The width of the rectangle where the contents will be arranged.
- **height** (*int, float*) – The height of the rectangle where the contents will be arranged.
- **pdf** (PDF, *optional*) – A PDF object used to get string styles inside the elements.

Raises **TypeError** – if `content` is not a dict

setup(*x=None*, *y=None*, *width=None*, *height=None*)

Function to change any or all of the parameters of the rectangle of the content.

Parameters

- **x** (*int, float, optional*) – The x coordinate of the left of the rectangle.
- **y** (*int, float, optional*) – The y coordinate of the top of the rectangle.
- **width** (*int, float, optional*) – The width of the rectangle where the contents will be arranged.
- **height** (*int, float, optional*) – The height of the rectangle where the contents will be arranged.

run(*x=None*, *y=None*, *width=None*, *height=None*)

Function to arrange this object elements in the rectangle defined by x, y, width and height.

More information about this method in this class definition.

Parameters

- **x** (*int, float, optional*) – The x position of the left of the rectangle.
- **y** (*int, float, optional*) – The y position of the top of the rectangle.
- **width** (*int, float, optional*) – The width of the rectangle where the contents will be arranged.
- **height** (*int, float, optional*) – The height of the rectangle where the contents will be arranged.

get_state()

Method to get the current state of this content box. This can be used later in method *pdfme.content.PDFContent.set_state()* to restore this state in this content box (like a checkpoint in a videogame).

Returns a dict with the state of this content box.

Return type dict

set_state(*section_element_index=None*, *section_delayed=None*, *children_memory=None*)

Method to set the state of this content box.

The 3 arguments of this method define the current state of this content box, and with this method you can change that state.

Parameters

- **section_element_index** (*int, optional*) – the index of the current element being added.
- **section_delayed** (*list, optional*) – a list of delayed elements, that should be added before continuing with the rest of elements.

- **children_memory** (*list, optional*) – if the current element is in turn a content box, this list says what the indexes of the nested content boxes inside this content box are.

class pdfme.content.**PDFContentPart**(*content*, *pdf_content*, *min_x*, *width*, *min_y*, *max_y*, *parent=None*, *last=False*, *inherited_style=None*)

Bases: object

Class that represent a content element.

This class has all the logic to arrange the content elements in the rectangle defined by min_x (left), min_y (top), width and max_y (bottom). This class needs a reference to a *pdfme.content.PDFContent* that will store the information of the lines, fills and parts of the elements arranged by this class, and all of the children PDFContentPart 's of this object. The description of the content argument is the same that the one from *pdfme.content.PDFContent*.

Parameters

- **content** (*dict*) – A content dict.
- **pdf_content** (PDFContent) – To store the fills, lines and parts of the elements of the content.
- **min_x** (*int, float*) – The x position of the left of the rectangle.
- **width** (*int, float*) – The width of the rectangle where the contents will be arranged.
- **min_y** (*int, float*) – The y position of the top of the rectangle.
- **max_y** (*int, float*) – The y position of the bottom of the rectangle.
- **parent** (PDFContentPart, *optional*) – If not None, this is the parent of the current object, and it's needed because the arranging process made by this object affects the parent arranging process and viceversa.
- **last** (*bool, optional*) – This tells whether this is the last element of the list of elements of the parent. Defaults to False.
- **inherited_style** (*dict, optional*) – The accumulated styles of all of the ancestors of the current object.

Raises **TypeError** – If content is not a dict

setup(*min_x*, *width*, *min_y*, *max_y*)

Function to update the rectangle of this element.

Parameters

- **min_x** (*int, float*) – The x position of the left of the rectangle.
- **width** (*int, float*) – The width of the rectangle where the contents will be arranged.
- **min_y** (*int, float*) – The y position of the top of the rectangle.
- **max_y** (*int, float*) – The y position of the bottom of the rectangle.

get_state()

Method to get the current state of this content box. This can be used later in method *pdfme.content.PDFContentPart.set_state()* to restore this state in this content box (like a checkpoint in a videogame).

Returns a dict with the state of this content box.

Return type dict

set_state(*section_element_index=None*, *section_delayed=None*, *children_memory=None*)

Method to set the state of this content box part.

The arguments of this method define the current state of this content box part, and with this method you can change that state.

Parameters

- **section_element_index** (*int, optional*) – the index of the current element being added.
- **section_delayed** (*list, optional*) – a list of delayed elements, that should be added before continuing with the rest of elements.
- **children_memory** (*list, optional*) – if the current element is in turn a content box, this list says what the indexes of the nested content boxes inside this content box are.

add_delayed()

Function to add the delayed elements to the rectangle.

This function will try to add the delayed elements to the rectangle and it will return a string telling what the main loop should do, depending on what happened with the elements when they were being added to the rectangle.

Returns any of the strings mentioned in *pdfme.content.PDFContentPart.add_elements()*.

add_elements()

Function to add the elements in content to the rectangle.

This function will try to add the elements to the rectangle and it will return a string telling what the main loop should do, depending on what happened with the elements when they were being added to the rectangle.

Returns

- `'interrupt'` means this element or one of its children reached the end of the rectangle of this element's root ancestor, or what is the same, this element's *pdfme.content.PDFContent* instance (the one saved in `pdf_content` attribute). This message will propagate to the ancestors until it reach the root ancestor and make the `pdf_content` to end running. After that the user should set a new rectangle, maybe in a new page, and call the *pdfme.content.PDFContent.run()* function again to keep adding the remaining elements that couldn't be added before.
- `'break'` means an ancestor is resetting and this element should stop adding elements.
- `'partial_next'` means an ancestor has some remaining elements that need to be added and this element should stop.
- `'next'` means that this element needs to move to the next section, to continue adding elements to the rectangle. The next section could be the next column of this element, or the next section of the parent.
- `'continue'` means this element is done adding all of the elements (there could be delayed elements still).

is_element_resetting()

Function that returns a string depending on whether this element is resetting or not.

Returns A string telling the main loop what should do next.

process_add_ans(*ans*)

Function that process the answers from methods *pdfme.content.PDFContentPart.add_delayed()* and *pdfme.content.PDFContentPart.add_elements()*.

Parameters **ans** (*str*) – Any of the strings described in *pdfme.content.PDFContentPart.add_elements()*.

Returns A string telling the main loop what should do next.

run()

Function to run the main loop that will add the content elements to the rectangle.

Returns A string telling the parent what should it do afterwards.

last_child_of_resetting()

Function that recursively, towards the ancestors, checks if this element is the last element of the last element of one ancestor that is resetting.

Returns True if this element is the last element of an ancestor that is resetting.

start_resetting()

Function that sets the attribute `will_reset` of this element or one of its ancestors to True.

reset()

Function that first checks if resetting process is over, and if not calculates a new value for attribute `max_y` and resets all of the elements added to the rectangle so far to repeat the arranging process.

Returns True if resetting process should continue or False if this process is done.

go_to_beginning()

Function that takes the x and y coordinates of this element to the `min_x` and `min_y` coordinates.

next_section(*children_memory=None*)

Function that sets the x and y position of this element in the next section.

The next section could be the next column of this element or the next section of one of the ancestors. If some ancestor is resetting, or the end of the rectangle of the root element is reached, a string with a instruction for the caller will propagate this message towards the ancestors to act according to it.

Parameters **children_memory** (*list, optional*) – This is a list containing the children's indexes and delayed elements, that is accumulated towards the ancestors.

Returns A string containing a message to the main loop, or a dict containing the new x and y coordinates that the children are going to have from now on.

Return type str, dict

get_min_x()

Function to get the x coordinate of the rectangle depending on the current column.

Returns The x coordinate.

Return type int, float

update_dimensions(*style*)

Function that updates the rectangle dimensions of the child element that is going to be added to the rectangle of this element.

Parameters **style** (*dict*) – The style dict that contains the margin information needed to calculate the child element rectangle dimensions.

`add_top_margin`(*style*)

Function that adds the top margin of the current child element.

Parameters `style` (*dict*) – The style dict that contains the margin information needed to calculate the child element top margin.

`parse_element`(*element*)

`process`(*element*, *last=False*)

Function to add a single child element to the rectangle.

This function will add an element to the rectangle, using the method corresponding to the type of the object (text, image, table or another content). Depending on what happens with the element (if it was added or delayed) this return a string with a message for the main loop, or a dict with information to tell the caller function what should be done afterwards.

Parameters

- `element` (*dict, str, list, tuple*) – The object representing the element to be added.
- `last` (*bool*) – Wheter or not this is the last child element of the list of child elements of this element.

`process_text`(*element*, *style*, *element_style*, *add_parts=True*, *add_top_margin=True*)

Function that tries to add a paragraph to the current column rectangle, and add the remainder to the delayed list

Parameters

- `element` (*dict*) – The paragraph to be added
- `style` (*dict*) – The style of the paragraph, combined with the style of this element.
- `element_style` (*dict*) – The style of the paragraph.

Returns Containing instructions to the caller.

Return type dict

`process_image`(*element*, *style*, *add_parts=True*, *add_top_margin=True*)

Function that tries to add an image to the current column rectangle, and add it to the delayed list if it can't add it.

Parameters

- `element` (*dict*) – The image to be added
- `style` (*dict*) – The style of the image.

Returns Containing instructions to the caller.

Return type dict

`process_table`(*element*, *style*, *element_style*, *add_parts=True*, *add_top_margin=True*)

Function that tries to add a table to the current column rectangle, and add the remainder to the delayed list.

Parameters

- `element` (*dict*) – The table to be added.
- `style` (*dict*) – The style of the table, combined with the style of this element.
- `element_style` (*dict*) – The style of the table.

Returns Containing instructions to the caller.

Return type dict

process_child(*element*, *style*, *last*, *add_top_margin=True*)

Function that tries to add a child content to the current column rectangle.

Parameters

- **element** (*dict*) – The child to be added
- **style** (*dict*) – The style of the child, combined with the style of this element.
- **last** (*bool*) – whether or not this is the last child of this element.

Returns Containing instructions to the caller.

Return type str, dict

get_element_styles(*element*, *inherited_style*)

process_group_element(*element*, *inherited_style*, *add_element=False*, *add_top_margin=True*, *min_height=None*)

process_group(*group_element*, *style*)

Function that tries to add a group element to the current column rectangle, and add it to the delayed list if it can't add it. If after 50 tries it can not add the group element, it will throw an exception.

Parameters

- **group_element** (*dict*) – The group element to be added
- **style** (*dict*) – The style of the group element, combined with the style of this content element.

Returns Containing instructions to the caller.

Return type dict

5.3.4 pdfme.document

class pdfme.document.**PDFDocument**(*document*, *context=None*)

Bases: object

Class that helps to build a PDF document from a dict (document argument) describing the document contents.

This class uses an instance of *pdfme.pdf.PDF* internally to build the PDF document, but adds some functionalities to allow the user to build a PDF document from a JSONish dict, add footnotes and other functions explained here.

A document is made up of sections, that can have their own page layout, page numbering, running sections and style.

document dict can have the following keys:

- style: the default style of each section inside the document. A dict with all of the keys that a content box can have (see *pdfme.content.PDFContent* for more information about content box, and for the default values of the attributes of this dict see *pdfme.pdf.PDF*). Additional to the keys of content box style, you can add the following keys: outlines_level, page_size, rotate_page, margin, page_numbering_offset and page_numbering_style. For more information about this page attributes and their default values see *pdfme.pdf.PDF* definition.
- formats: a dict with the global styles of the document that can be used anywhere in the document. For more information about this dict see *pdfme.pdf.PDF* definition.

- `running_sections`: a dict with the running sections that will be used by each section in the document. Each section can have, in turn, a `running_section` list, with the name of the running sections defined in this argument that should be included in the section. For information about running sections see *pdfme.pdf.PDF*. If `width` key is equal to `'left'`, it takes the value of the left margin, if equal to `'right'` it takes the value of the right margin, if equal to `'full'` it takes the value of the whole page width, and if it is not defined or is None it will take the value of the content width of the page. If `height` key is equal to `'top'`, it takes the value of the top margin, if equal to `'bottom'` it takes the value of the bottom margin, if equal to `'full'` it takes the value of the whole page height, and if it is not defined or is None it will take the value the content height of the page. If `x` key is equal to `'left'`, it takes the value of the left margin, if equal to `'right'` it takes the value of the whole page width minus the right margin, and if it is not defined or is None it will be 0. If `y` key is equal to `'top'`, it takes the value of the top margin, if equal to `'bottom'` it takes the value of the whole page height minus the bottom margin, and if it is not defined or is None i will be 0.
- `per_page`: a list of dicts, each with a mandatory key `pages`, a comma separated string of indexes or ranges (python style), and any of the following optional keys:
  - `style`: a style dict with page related style properties (page_size, rotate_page, margin) that will be applied to every page in the `pages` ranges.
  - `running_sections`: a dict with optional `exclude` and `include` lists of running sections names to be included and excluded in every page in the `pages` ranges.
- `sections`: an iterable with the sections of the document.

Each section in `sections` iterable is a dict like the one that can be passed to *pdfme.content.PDFContent*, so each section ends up being a content box. This class will add as many pages as it is needed to add all the contents of every section (content box) to the PDF document.

Additional to the keys from a content box dict, you can include a `running_sections` list with the name of the running sections that you want to be included in all of the pages of the section. There is a special key that you can include in a section's `style` dict called `page_numbering_reset`, that if True, resets the numbering of the pages.

You can also include footnotes in any paragraph, by adding a dict with the key `footnote` with the description of the footnote as its value, to the list of elements of the dot key (see *pdfme.text.PDFText* for more informarion about the structure of a paragraph and the dot key).

Here is an example of a document dict, and how it can be used to build a PDF document using the helper function *pdfme.document.build_pdf()*.

```
from pdfme import build_pdf

document = {
    "style": {
        "page_size": "letter", "margin": [70, 60],
        "s": 10, "c": 0.3, "f": "Times", "text_align": "j",
        "margin_bottom": 10
    },
    "formats": {
        "link": {"c": "blue", "u": True},
        "title": {"s": 12, "b": True}
    },
    "running_sections": {
        "header": {
            "x": "left", "y": 40, "height": "top",
            "content": ["Document with header"]
```

(continues on next page)

(continued from previous page)

```
        },
        "footer": {
            "x": "left", "y": "bottom", "height": "bottom",
            "style": {"text_align": "c"},
            "content": [{".": ["Page ", {"var": "$page"}]}]
        }
    },
    "sections": [
        {
            "running_sections": ["header", "footer"],
            "style": {"margin": 60},
            "content": [
                {".": "This is a title", "style": "title"},
                {".": [
                    "Here we include a footnote",
                    {"footnote": "Description of a footnote"},
                    ". And here we include a ",
                    {
                        ".": "link", "style": "link",
                        "uri": "https://some.url.com"
                    }
                ]}
            ]
        },
        {
            "running_sections": ["footer"],
            "style": {"rotate_page": True},
            "content": [
                "This is a rotated page"
            ]
        }
    ]
}

with open('document.pdf', 'wb') as f:
    build_pdf(document, f)
```

Parameters

- **document** (*dict*) – a dict like the one just described.
- **context** (*dict, optional*) – a dict containing the context of the inner *pdfme.pdf.PDF* instance.

run()

Method to process this document sections.

output(*buffer*)

Method to create the PDF file.

Parameters **buffer** (*file_like*) – a file-like object to write the PDF file into.

pdfme.document.**build_pdf**(*document*, *buffer*, *context=None*)

Function to build a PDF document using a PDFDocument instance. This is the easiest way to build a PDF docu-

ment file in this library. For more information about arguments `document`, and `context` see *pdfme.document.PDFDocument*.

Parameters **buffer** (*file_like*) – a file-like object to write the PDF file into.

5.3.5 pdfme.encoders

pdfme.encoders.**encode_stream**(*stream*, *filter*, *parameters=None*)

Function to use `filter` method to encode `stream`, using `parameters` if required.

Parameters

- **stream** (*bytes*) – the stream to be encoded.
- **filter** (*bytes*) – the method to use for the encoding process.
- **parameters** (*dict, optional*) – if necessary, this dict contains the parameters required by the `filter` method.

Raises

- **NotImplementedError** – if the filter passed is not implemented yet.
- **Exception** – if the filter passed doesn't exist.

Returns the encoded stream.

Return type bytes

pdfme.encoders.**flate_encode**(*stream*)

Function that encodes a bytes stream using the zlib.compress method.

Parameters **stream** (*bytes*) – stream to be encoded.

Returns the encoded stream.

Return type bytes

5.3.6 pdfme.fonts

class pdfme.fonts.**PDFFont**(*ref*)

Bases: `abc.ABC`

Abstract class that represents a PDF font.

Parameters **ref** (*str*) – the name of the font, included in every paragraph and page that uses this font.

property ref

Property that returns the name (ref) of this font.

Returns the name of this font

Return type str

abstract property base_font

Abstract property that should return this font's base font name.

Returns the base font name

Return type str

abstract get_char_width(*char*)

Abstract method that should return the width of `char` character in this font.

Parameters **char** (*str*) – the character.

Returns the character's width.

Return type float

abstract get_text_width(*text*)

Abstract method that should return the width of the `text` string in this font.

Parameters **text** (*str*) – the sentence to measure.

Returns the sentence's width.

Return type float

abstract add_font(*base*)

Abstract method that should add this font to the PDFBase instance, passed as argument.

Parameters **base** (PDFBase) – the base instance to add this font.

class pdfme.fonts.**PDFStandardFont**(*ref*, *base_font*, *widths*)

Bases: *pdfme.fonts.PDFFont*

This class represents a standard PDF font.

Parameters

- **ref** (*str*) – the name of this font.
- **base_font** (*str*) – the base font name of this font.
- **widths** (*dict*) – the widths of each character in this font.

property base_font

See *pdfme.fonts.PDFFont.base_font()*

get_char_width(*char*)

See *pdfme.fonts.PDFFont.get_char_width()*

get_text_width(*text*)

See *pdfme.fonts.PDFFont.get_text_width()*

add_font(*base*)

See *pdfme.fonts.PDFFont.add_font()*

class pdfme.fonts.**PDFTrueTypeFont**(*ref*, *filename=None*)

Bases: *pdfme.fonts.PDFFont*

This class represents a TrueType PDF font.

This class is not working yet.

Parameters

- **ref** (*str*) – the name of this font.
- **base_font** (*str*) – the base font name of this font.
- **widths** (*dict*) – the widths of each character in this font.

property base_font

See *pdfme.fonts.PDFFont.base_font()*

get_char_width(*char*)

See *pdfme.fonts.PDFFont.get_char_width()*

get_text_width(*text*)

See *pdfme.fonts.PDFFont.get_text_width()*

load_font(*filename*)

Method to extract information needed by the PDF document about this font, from font file in `filename` path.

Parameters **filename** (*str*) – font file path.

Raises **ImportError** – if `fonttools` library is not installed.

add_font(*base*)

See *pdfme.fonts.PDFFont.add_font()*

class pdfme.fonts.**PDFFonts**

Bases: `object`

Class that represents the set of all the fonts added to a PDF document.

get_font(*font_family*, *mode*)

Method to get a font from its `font_family` and `mode`.

Parameters

- **font_family** (*str*) – the name of the font family
- **mode** (*str*) – the mode of the font you want to get. `n`, `b`, `i` or `bi`.

Returns an object that represents a PDF font.

Return type *PDFFont*

load_font(*path*, *font_family*, *mode='n'*)

Method to add a TrueType font to this instance.

Parameters

- **path** (*str*) – the location of the font file.
- **font_family** (*str*) – the name of the font family
- **mode** (*str, optional*) – the mode of the font you want to get. `n`, `b`, `i` or `bi`.

5.3.7 pdfme.image

class pdfme.image.**PDFImage**(*image*, *extension=None*, *image_name=None*)

Bases: `object`

Class that represents a PDF image.

You can pass the location path (`str` or `pathlib.Path` format) of the image, or pass a file-like object (`io.BufferedIOBase`) with the image bytes, the extension of the image, and the image name.

Only JPEG and PNG image formats are supported in this moment. PNG images are converted to JPEG, and for this Pillow library is required.

Parameters

- **image** (*str, pathlib.Path, BufferedIOBase*) – The path or file-like object of the image.

- **extension** (*str, optional*) – If `image` is path-like object, this argument should contain the extension of the image. Options are [`jpg`, `jpeg`, `png`].
- **image_name** (*str, optional*) – If `image` is path-like object, this argument should contain the name of the image. This name should be unique among the images added to the same PDF document.

parse_jpg(*bytes_*)

Method to extract metadata from a JPEG image `bytes_` needed to embed this image in a PDF document.

This method creates this instance's attibute `pdf_obj`, containing a dict that can be added to a *pdfme.base.PDFBase* instance as a PDF Stream object that represents this image.

Parameters **bytes** (*BufferedIOBase*) – A file-like object containing the image.

parse_png(*bytes_*)

Method to convert a PNG image to a JPEG image and later parse it as a JPEG image.

This method creates this instance's attibute `pdf_obj`, containing a dict that can be added to a *pdfme.base.PDFBase* instance as a PDF Stream object that represents this image.

Parameters **bytes** (*BinaryIO*) – A file-like object containing the image.

5.3.8 pdfme.page

class pdfme.page.**PDFPage**(*base*, *width*, *height*, *margin_top=0*, *margin_bottom=0*, *margin_left=0*, *margin_right=0*)

Bases: `object`

Class that represents a PDF page, and has methods to add stream parts into the internal page PDF Stream Object, and other things like fonts, annotations and images.

This object have `x` and `y` coordinates used by the *pdfme.pdf.PDF* insance that contains this page. This point is called `cursor` in this class.

Parameters

- **base** (*PDFBase*) – [description]
- **width** (*Number*) – [description]
- **height** (*Number*) – [description]
- **margin_top** (*Number, optional*) – [description]. Defaults to 0.
- **margin_bottom** (*Number, optional*) – [description]. Defaults to 0.
- **margin_left** (*Number, optional*) – [description]. Defaults to 0.
- **margin_right** (*Number, optional*) – [description]. Defaults to 0.

property y

Returns The current vertical position of the page's cursor, from top (0) to bottom. This is different from _y attribute, the position from bottom (0) to top.

Return type Number

go_to_beginning()

Method to set the position of the cursor's page to the origin point of the page, considering this page margins. The origin is at the left-top corner of the rectangle that will contain the page's contents.

add(*content*)

Method to add some bytes (if a string is passed, it's transformed into a bytes object) representing a stream portion, into this page's PDF internal Stream Object.

Parameters **content** (*str, bytes*) – the stream portion to be added to this page's stream.

Returns the id of the portion added to the page's stream

Return type int

add_font(*font_ref*, *font_obj_id*)

Method to reference a PDF font in this page, that will be used inside this page's stream.

Parameters

- **font_ref** (*str*) – the `ref` attribute of the *pdfme.fonts.PDFFont* instance that will be referenced in this page.
- **font_obj_id** (PDFRef) – the object id of the font being referenced here, already added to a *pdfme.base.PDFBase* instance.

add_annot(*obj*, *rect*)

Method to add a PDF annotation to this page.

The `object` dict should have the keys describing the annotation to be added. By default, this object will have the following key/values by default: `Type = /Annot` and `Subtype = /Link`. You can include these keys in `object` if you want to overwrite any of the default values for them.

Parameters

- **obj** (*dict*) – the annotation object.
- **rect** (*list*) – a list with the following information about the annotation: [x, y, width, height].

add_link(*uri_id*, *rect*)

Method to add a link annotation (a URI that opens a webpage from the PDF document) to this page.

Parameters

- **uri_id** (PDFRef) – the object id of the action object created to open this link.
- **rect** (*list*) – a list with the following information about the annotation: [x, y, width, height].

add_reference(*dest*, *rect*)

Method to add a reference annotation (a clickable area, that takes the user to a destination) to this page.

Parameters

- **dest** (*str*) – the name of the dest being referenced.
- **rect** (*list*) – a list with the following information about the annotation: [x, y, width, height].

add_image(*image_obj_id*, *width*, *height*)

Method to add an image to this page.

The position of the image will be the same as `x` and `y` coordinates of this page.

Parameters

- **image_obj_id** (PDFRef) – the object id of the image PDF object.
- **width** (*int, float*) – the width of the image.

- **height** (*int, float*) – the height of the image.

5.3.9 pdfme.parser

class pdfme.parser.**PDFObject**(*id_*, *obj=None*)

Bases: object

A class that represents a PDF object.

This object has a *pdfme.parser.PDFRef* id attribute representing the id of this object inside the PDF document, and acts as a dict, so the user can update any property of this PDF object like you would do with a dict.

Parameters

- **id** (PDFRef) – The id of this object inside the PDF document.
- **obj** (*dict, optional*) – the dict representing the PDF object.

class pdfme.parser.**PDFRef**(*id_*)

Bases: int

An int representing the id of a PDF object.

This is a regular int that has an additional property called ref with a representation of this object, to be referenced elsewhere in the PDF document.

property ref

Returns bytes with a representation of this object, to be referenced elsewhere in the PDF document.

Return type bytes

pdfme.parser.**parse_obj**(*obj*)

Function to convert a python object to a bytes object representing the corresponding PDF object.

Parameters **float,** (*obj (PDFObject, PDFRef, dict, list, tuple, set, bytes, bool, int,*) – str): the object to be converted to a PDF object.

Returns bytes representing the corresponding PDF object.

Return type bytes

pdfme.parser.**parse_dict**(*obj*)

Function to convert a python dict to a bytes object representing the corresponding PDF Dictionary.

Parameters **obj** (*dict*) – the dict to be converted to a PDF Dictionary.

Returns bytes representing the corresponding PDF Dictionary.

Return type bytes

pdfme.parser.**parse_list**(*obj*)

Function to convert a python iterable to a bytes object representing the corresponding PDF Array.

Parameters **obj** (*iterable*) – the iterable to be converted to a PDF Array.

Returns bytes representing the corresponding PDF Array.

Return type bytes

pdfme.parser.**parse_stream**(*obj*)

Function to convert a dict representing a PDF Stream object to a bytes object.

A dict representing a PDF stream should have a `'__stream__` key containing the stream bytes. You don't have to include `Length` key in the dict, as it is calculated by us. The value of `'__stream__'` key must be of type `bytes` or a dict whose values are of type `bytes`. If you include a `Filter` key, a encoding is automatically done in the stream (see *pdfme.encoders.encode_stream()* function for supported encoders). If the contents of the stream are already encoded using the filter in `Filter` key, you can skip the encoding process by including the `__skip_filter__` key.

Parameters **obj** (*dict*) – the dict representing a PDF stream.

Returns bytes representing the corresponding PDF Stream.

Return type bytes

5.3.10 pdfme.pdf

class pdfme.pdf.**PDF**(*page_size='a4'*, *rotate_page=False*, *margin=56.693*, *page_numbering_offset=0*, *page_numbering_style='arabic'*, *font_family='Helvetica'*, *font_size=11*, *font_color=0.1*, *text_align='l'*, *line_height=1.1*, *indent=0*, *outlines_level=1*)

Bases: `object`

Class that represents a PDF document, and has methods to add pages, and to add paragraphs, images, tables and a mix of this, a content box, to them.

You can use this class to create a PDF file, by adding one page at a time, and adding stuff to each page you add, like this:

```
from pdfme import PDF

pdf = PDF()
pdf.add_page()
pdf.text('This is a paragraph')

with open('document.pdf', 'wb') as f:
    pdf.output(f)
```

Through the constructor arguments you can modify the default features of the PDF document, like the size of the pages, their orientation, the page numbering options, and the appearance of the text. These are used everytime you create a new page, or a new paragraph, but you can overwrite these for each case.

You can change the default values for the pages by calling *pdfme.pdf.PDF.setup_page()*, and change the default values for text by changing attributes `font_family`, `font_size`, `font_color`, `text_align` and `line_height`.

Methods *pdfme.pdf.PDF.text()*, *pdfme.pdf.PDF.image()*, *pdfme.pdf.PDF.table()* and *pdfme.pdf.PDF.content()* are the main functions to add paragraphs, images, tables and content boxes respectively, and all of them, except the image method, take into account the margins of the current page you are working on, and create new pages automatically if the stuff you are adding needs more than one page. If you want to be specific about the position and the size of the paragraphs, tables and content boxes you are inserting, you can use methods *pdfme.pdf.PDF._text()*, *pdfme.pdf.PDF._table()* and *pdfme.pdf.PDF._content()* instead, but these don't handle the creation of new pages like the first ones.

Each page has attributes `x` and `y` that are used to place elements inside them, and for the methods that receive `x` and `y` arguments, if they are None, the page's `x` and `y` attributes are used instead.

For more information about paragraphs see *pdfme.text.PDFText*, and about tables *pdfme.table.PDFTable*.

Although you can add all of the elements explained so far, we recommend using content boxes only, because all of the additional funcionalities they have, including its ability to embed other elements. For more information about content boxes see *pdfme.content.PDFContent*.

Paragraphs, tables and content boxes use styles to give format to the content inside of them, and sometimes styling can get repetitive. This is why there's a dict attribute called `formats` where you can add named style dicts and used them everywhere inside this document, like this:

```
from pdfme import PDF

pdf = PDF()
pdf.formats['link'] = {
    'c': 'blue',
    'u': True
}
pdf.add_page()
pdf.text({
    '.': 'this is a link',
    'style': 'link',
    'uri': 'https://some.domain.com'
})
```

If you find yourself using a piece of text often in the document, you can add it to the dict attribute `context` and include it in any paragraph in the document by using its key in the dict, like this:

```
from pdfme import PDF

pdf = PDF()
pdf.context['arln'] = 'A Really Long Name'
pdf.add_page()
pdf.text({
    '.': ['The following name is ', {'var': 'arln'}, '.']
})
```

There are some special `context` variables that are used by us that start with symbol `$`, so it's adviced to name your own variables without this symbol in the beginning. The only of these variables you should care about is `$page` that contains the number of the current page.

You can add as much running sections as you want by using *pdfme.pdf.PDF.add_running_section()*. Running sections are content boxes that are included on every page you create after adding them. Through these you can add a header and a footer to the PDF.

If you want a simpler and more powerful interface, you should use *pdfme.document.PDFDocument*.

Parameters

- **page_size** (*str, int, float, tuple, list, optional*) – this argument sets the dimensions of the page. See *pdfme.utils.get_page_size()*.
- **rotate_page** (*bool, optional*) – whether the page dimensions should be inverted (True), or not (False).
- **margin** (*str, int, float, tuple, list, dict, optional*) – the margins of the pages. See *pdfme.utils.parse_margin()*.
- **page_numbering_offset** (*int, float, optional*) – if the number of the page is included, this argument will set the offset of the page. For example if the current page is the 4th one, and the offset is 3, the page number displayed in the current page will be 1.

- **page_numbering_style** (*str, optional*) – the style of the page number. Options are `arabic` (1,2,3,...) and `roman` (I, II, III, IV, ...).
- **font_family** (*str, optional*) – The name of the font family. Options are `Helvetica` (default), `Times`, `Courier`, `Symbol` and `ZapfDingbats`. You will also be able to add new fonts in a future release.
- **font_size** (*in, optional*) – The size of the font.
- **font_color** (*int, float, str, list, tuple, optional*) – The color of the font. See *pdfme.color.parse_color()*.
- **text_align** (*str, optional*) – `'l'` for left (default), `'c'` for center, `'r'` for right and `'j'` for justified text.
- **line_height** (*int, float, optional*) – space between the lines of the paragraph. See *pdfme.text.PDFText*.
- **indent** (*int, float, optional*) – space between left of the paragraph, and the beggining of the first line. See *pdfme.text.PDFText*.
- **outlines_level** (*int, optional*) – the level of the outlines to be displayed on the outlines panel when the PDF document is opened.

property page

Returns current page

Return type *PDFPage*

property page_index

Returns current page index.

Return type int

property width

Returns current page width

Return type float

property height

Returns current page height

Return type float

setup_page(*page_size=None*, *rotate_page=None*, *margin=None*)

Method to set the page features defaults. These values will be used from now on when adding new pages.

Parameters

- **page_size** (*str, int, float, tuple, list, optional*) – this argument sets the dimensions of the page. See *pdfme.utils.get_page_size()*.
- **rotate_page** (*bool, optional*) – whether the page dimensions should be inverted (True), or not (False).
- **margin** (*str, int, float, tuple, list, dict, optional*) – the margins of the pages. See *pdfme.utils.parse_margin()*.

add_page(*page_size=None*, *rotate_page=None*, *margin=None*)

Method to add a new page. If provided, arguments will only apply for the page being added.

Parameters

- **page_size** (*str, int, float, tuple, list, optional*) – this argument sets the dimensions of the page. See *pdfme.utils.get_page_size()*.
- **rotate_page** (*bool, optional*) – whether the page dimensions should be inverted (True), or not (False).
- **margin** (*str, int, float, tuple, list, dict, optional*) – the margins of the page. See *pdfme.utils.parse_margin()*.

add_running_section(*content*, *width*, *height*, *x*, *y*)

Method to add running sections, like a header and a footer, to this document.

Running sections are content boxes that are included on every page you create after adding them.

Parameters

- **content** (*dict*) – a content dict like the one you pass to create a instance of *pdfme.content.PDFContent*.
- **width** (*int, float, optional*) – The width of the rectangle where the contents will be arranged.
- **height** (*int, float, optional*) – The height of the rectangle where the contents will be arranged.
- **x** (*int, float, optional*) – The x position of the left of the rectangle.
- **y** (*int, float, optional*) – The y position of the top of the rectangle.

add_font(*fontfile*, *font_family*, *mode='n'*)

Method to add a new font to this document. This functionality is not ready yet.

Parameters

- **fontfile** (*str*) – the path of the fontfile.
- **font_family** (*str*) – the name of the font family being added.
- **mode** (*str, optional*) – the mode of the font being added. It can be `n` for normal, `b` for bold and `i` for italics (oblique).

create_image(*image*, *extension=None*, *image_name=None*)

Method to create a PDF image.

Arguments for this method are the same as *pdfme.image.PDFImage*.

Returns object representing the PDF image.

Return type *PDFImage*

add_image(*pdf_image*, *x=None*, *y=None*, *width=None*, *height=None*, *move='bottom'*)

Method to add a PDF image to the current page.

Parameters

- **pdf_image** (PDFImage) – the PDF image.
- **x** (*int, float, optional*) – The x position of the left of the image.
- **y** (*int, float, optional*) – The y position of the top of the image.

- **width** (*int, float, optional*) – The width of the image. If this and `height` are None, the width will be the same as the page content width, but if this is None and `height` is not, the width will be calculated from `height`, keeping the proportion of the image.
- **height** (*int, float, optional*) – The height of the image. If this is None, the height will be calculated from the image `width`, keeping the proportion.
- **move** (*str, optional*) – wheter it should move page x coordinate to the right side of the image (`next`) or if it should move page y coordinate to the bottom of the image (`bottom`) (default).

image(*image*, *extension=None*, *image_name=None*, *x=None*, *y=None*, *width=None*, *height=None*, *move='bottom'*)

Method to create and add a PDF image to the current page.

Parameters

- **image** (*str, Path, BytesIO*) – see *pdfme.image.PDFImage*.
- **extension** (*str, optional*) – see *pdfme.image.PDFImage*.
- **image_name** (*str, optional*) – see *pdfme.image.PDFImage*.
- **x** (*int, float, optional*) – the x position of the left of the image.
- **y** (*int, float, optional*) – the y position of the top of the image.
- **width** (*int, float, optional*) – the width of the image. If this and `height` are None, the width will be the same as the page content width, but if this is None and `height` is not, the width will be calculated from `height`, keeping the proportion of the image.
- **height** (*int, float, optional*) – the height of the image. If this is None, the height will be calculated from the image `width`, keeping the proportion.
- **move** (*str, optional*) – wheter it should move page x coordinate to the right side of the image (`next`) or if it should move page y coordinate to the bottom of the image (`bottom`) (default).

get_page_number()

Method that returns the string reprensentation of the number of the current page.

Returns string with the page number that depends on attributes `page_numbering_offset` and `page_numbering_style`.

Return type str

_text(*content*, *width=None*, *height=None*, *x=None*, *y=None*, *text_align=None*, *line_height=1.1*, *indent=0*, *list_text=None*, *list_indent=None*, *list_style=None*, *move='bottom'*)

Method to create and add a paragraph to the current page.

If `content` is a PDFText, the method `run` for this instance will be called with the new rectangle passed to this function. Else, this method will try to build a new PDFText instance with argument `content` and call method `run` afterwards.

For more information about the arguments see *pdfme.text.PDFText*.

Returns object that represents the paragraph.

Return type *PDFText*

text(*content*, *text_align=None*, *line_height=1.1*, *indent=0*, *list_text=None*, *list_indent=None*, *list_style=None*)

Method to create and add a paragraph to this document. This method will keep adding pages to the PDF until all the contents of the paragraph are added to the document.

For more information about the arguments see *pdfme.text.PDFText*.

_table(*content*, *width=None*, *height=None*, *x=None*, *y=None*, *widths=None*, *style=None*, *borders=None*, *fills=None*, *move='bottom'*)

Method to create and add a table to the current page.

If `content` is a PDFTable, the method `run` for this instance will be called with the new rectangle passed to this function. Else, this method will try to build a new PDFTable instance with argument `content` and call method `run` afterwards.

For more information about this method arguments see *pdfme.table.PDFTable*.

Returns object that represents a table.

Return type *PDFTable*

table(*content*, *widths=None*, *style=None*, *borders=None*, *fills=None*)

Method to create and add a table to this document. This method will keep adding pages to the PDF until all the contents of the table are added to the document.

For more information about this method arguments see *pdfme.table.PDFTable*.

_content(*content*, *width=None*, *height=None*, *x=None*, *y=None*, *move='bottom'*)

Method to create and add a content box to teh current page.

If `content` is a PDFContent, the method `run` for this instance will be called with the new rectangle passed to this function. Else, this method will try to build a new PDFContent instance with argument `content` and call method `run` afterwards.

For more information about this method arguments see *pdfme.content.PDFContent*.

Returns object that represents a content box.

Return type *PDFContent*

content(*content*)

Method to create and add a content box to this document. This method will keep adding pages to the PDF until all the contents are added to the document.

Parameters **content** (*dict*) – see *pdfme.content.PDFContent*.

output(*buffer*)

Method to create the PDF file.

Parameters **buffer** (*file_like*) – a file-like object to write the PDF file into.

Raises **Exception** – if this document doesn't have any pages.

5.3.11 pdfme.table

class pdfme.table.**PDFTable**(*content*, *fonts*, *x*, *y*, *width*, *height*, *widths=None*, *style=None*, *borders=None*, *fills=None*, *pdf=None*)

Bases: `object`

Class that represents a PDF table.

The `content` argument is an iterable representing the rows of the table, and each row should be an iterable too, representing each of the columns in the row. The elements on a row iterable could be any of the elements that you pass to argument `content` list in class *pdfme.content.PDFContent*. Because of this you can add paragraphs, images and content boxes into a table cell.

Argument `widths`, if passed, should be an iterable with the width of each column in the table. If not passed, all the columns will have the same width.

Argument `style`, if passed, should be a dict with any of the following keys:

- `cell_margin`: the margin of the four sides of the cells in the table. Default value is `5`.
- `cell_margin_left`: left margin of the cells in the table. Default value is `cell_margin`.
- `cell_margin_top`: top margin of the cells in the table. Default value is `cell_margin`.
- `cell_margin_right`: right margin of the cells in the table. Default value is `cell_margin`.
- `cell_margin_bottom`: bottom margin of the cells in the table. Default value is `cell_margin`.
- `cell_fill`: the color of all the cells in the table. Default value is `None` (transparent). See *pdfme.color.parse_color()* for information about this attribute.
- `border_width`: the width of all the borders in the table. Default value is `0.5`.
- `border_color`: the color of all the borders in the table .Default value is `'black'`. See *pdfme.color.parse_color()* for information about this attribute.
- `border_style`: the style of all the borders in the table. It can be `solid`, `dotted` or `solid`. Default value is `solid`.

You can overwrite the default values for the cell fills and the borders with `fills` and `borders` arguments. These arguments, if passed, should be iterables of dicts. Each dict should have a `pos` key that contains a string with information of the vertical (rows) and horizontal (columns) position of the fills or borders you want to change, and for this, such a string should has 2 parts separated by a semi colon, the first one for the vertical position and the second one for the horizontal position. The position can be a single int, a comma-separated list of ints, or a slice (range), like the one you pass to get a slice of a python list. For borders you have to include a `h` or a `v` before the positions, to tell if you want to change vertical or horizontal borders. The indexes in this string can be negative, referring to positions from the end to the beginning.

The following are examples of valid `pos` strings:

- `'h0,1,-1;:'` to modify the first, second and last horizontal lines in the table. The horizontal position is a single colon, and thus the whole horizontal lines are affected.
- `'::2;:'` to modify all of the fills horizontally, every two rows. This would set the current fill to all the cells in the first row, the third row, the fifth row and so on.

Additional to the `pos` key for dicts inside `fills` iterable, you have to include a `color` key, with a valid color value. See *pdfme.color.parse_color()* for information about this attribute.

Additional to the `pos` key for dicts inside `borders` iterable, you can include `width` (border width), `color` (border color) and `style` (border style) keys.

If a cell element is a dict it's `style` dict can have any of the following keys: `cell_margin`, `cell_margin_left`, `cell_margin_top`, `cell_margin_right`, `cell_margin_bottom` and `cell_fill`, to overwrite the default value of any of these parameters on its cell. In a cell dict, you can also include `colspan` and `rowspan` keys, to span it horizontally and vertically respectively. The cells being merged to this spanned cell should be None.

Here's an example of a valid `content` value:

```
[
    ['row 1, col 1', 'row 1, col 2', 'row 1, col 3'],
    [
        'row2 col1',
        {
            'style': {'cell_margin': 10, }
            'colspan': 2, 'rowspan': 2
            '.': 'rows 2 to 3, cols 2 to 3',
        },
        None
    ],
    ['row 3, col 1', None, None],
]
```

Use method *pdfme.table.PDFTable.run()* to add as many rows as possible to the rectangle defined by `x`, `y`, `width` and `height`. The rows are added to this rectangle, until they are all inside of it, or until all of the vertical space is used and the rest of the rows can not be added. In these two cases method `run` finishes, and the property `finished` will be True if all the elements were added, and False if the vertical space ran out. If `finished` is False, you can set a new rectangle (on a new page for example) and use method `run` again (passing the parameters of the new rectangle) to add the remaining elements that couldn't be added in the last rectangle. You can keep doing this until all of the elements are added and therefore property `finished` is True.

By using this method the rows are not really added to the PDF object. After calling `run`, the properties `fills` and `lines` will be populated with the fills and lines of the tables that fitted inside the rectangle, and `parts` will be filled with the paragraphs and images that fitted inside the table rectangle too, and you have to add them by yourself to the PDF object before using method `run` again (in case `finished` is False), because they will be redefined for the next rectangle after calling it again. You can check the `table` method in PDF module to see how this process is done.

Parameters

- **content** (*iterable*) – like the one just explained.
- **fonts** (PDFFonts) – a PDFFonts object used to build paragraphs.
- **x** (*int, float*) – the x position of the left of the table.
- **y** (*int, float*) – the y position of the top of the table.
- **width** (*int, float*) – the width of the table.
- **height** (*int, float*) – the height of the table.
- **widths** (*Iterable, optional*) – the widths of each column.
- **style** (*Union[dict, str], optional*) – the default style of the table.
- **borders** (*Iterable, optional*) – borders of the table.
- **fills** (*Iterable, optional*) – fills of the table.
- **pdf** (PDF, *optional*) – A PDF object used to get string styles inside the elements.

setup(*x=None*, *y=None*, *width=None*, *height=None*)

Method to change the size and position of the table.

Parameters

- **x** (*int, float, optional*) – the x position of the left of the table.
- **y** (*int, float, optional*) – the y position of the top of the table.
- **width** (*int, float, optional*) – the width of the table.
- **height** (*int, float, optional*) – the height of the table.

get_state()

Method to get the current state of this table. This can be used later in method *pdfme.table.PDFTable.set_state()* to restore this state in this table (like a checkpoint in a videogame).

Returns a dict with the state of this table.

Return type dict

set_state(*current_index=None*, *delayed=None*)

Method to set the state of this table.

The arguments of this method define the current state of this table, and with this method you can change that state.

Parameters

- **current_index** (*int, optional*) – the index of the current row being added.
- **delayed** (*dict, optional*) – a dict with delayed cells that should be added before the next row.

set_default_border()

Method to create attribute `default_border` containing the default border values.

parse_pos_string(*pos*, *counts*)

Method to convert a position string like the ones used in `borders` and `fills` arguments of this class, into a generator of positions obtained from this string.

For more information, see the definition of this class.

Parameters

- **pos** (*str*) – position string.
- **counts** (*int*) – the amount of columns or rows.

Yields *tuple* – the horizontal and vertical index of each position obtained from the `pos` string.

parse_range_string(*data*, *count*)

Method to convert one of the parts of a position string like the ones used in `borders` and `fills` arguments of this class, into a iterator with all the positions obtained from this string.

For more information, see the definition of this class.

Parameters

- **data** (*str*) – one of the parts of a position string.
- **counts** (*int*) – the amount of columns or rows.

Returns a list of indexes, or a `range` object.

Return type iterable

setup_borders(*borders*)

Method to process the borders argument passed to this class, and populate attributes borders_h and borders_v.

Parameters **borders** (*iterable*) – the borders argument passed to this class.

get_border(*i*, *j*, *is_vert*)

Method to get the border in the horizontal position i, and vertical position j. It takes a vertical border if is_vert is true, and a horizontal border if is_vert is false :param i: horizontal position. :type i: int :param j: vertical position. :type j: int :param is_vert: vertical (True) or horizontal (False) border. :type is_vert: bool

Returns dict with description of the border in position i, j.

Return type dict

setup_fills(*fills*)

Method to process the fills argument passed to this class, and populate attribute fills_defs.

Parameters **fills** (*iterable*) – the fills argument passed to this class.

compare_borders(*a*, *b*)

Method that compares border dicts a and b and returns if they are equal (True) or not (False)

Parameters

- **a** (*dict*) – first border.
- **b** (*dict*) – second border.

Returns if a and b are equal (True) or not (False).

Return type bool

process_borders(*col*, *border_left*, *border_top*)

Method to setup the top and left borders of each cell

Parameters

- **col** (*int*) – the columns number.
- **border_left** (*dict*) – the left border dict.
- **border_top** (*dict*) – the top border dict.

run(*x=None*, *y=None*, *width=None*, *height=None*)

Method to add as many rows as possible to the rectangle defined by x, y`, width and height attributes.

More information about this method in this class definition.

Parameters

- **x** (*int, float, optional*) – the x position of the left of the table.
- **y** (*int, float, optional*) – the y position of the top of the table.
- **width** (*int, float, optional*) – the width of the table.
- **height** (*int, float, optional*) – the height of the table.

add_row(*row*, *is_delayed=False*)

Method to add a row to this table.

Parameters

- **row** (*iterable*) – the row iterable.

- **is_delayed** (*bool, optional*) – whether this row is being added in delayed mode (`True`) or not (`False`).

Returns string with the action that should be performed after this row is added.

Return type str

get_cell_dimensions(*col*, *border_left*, *border_top*, *cell_style*, *rowspan*, *colspan*)

Method to get the cell dimensions at column `col`, taking into account the cell borders, and the column and row spans.

Parameters

- **col** (*int*) – the column of the cell.
- **border_left** (*dict*) – left border dict.
- **border_top** (*dict*) – top border dict.
- **cell_style** (*dict*) – cell style dict.
- **rowspan** (*int*) – the row span.
- **colspan** (*int*) – the column span.

Returns tuple with position `(x, y)`, size `(width, height)`, and padding `(left, top)` for this cell.

Return type tuple

is_span(*col*, *border_left*, *border_top*, *is_delayed*)

Method to check if cell at column `col` is part of a spanned cell (`True`) or not (`False`).

Parameters

- **col** (*int*) – the column of the cell.
- **border_left** (*dict*) – left border dict.
- **border_top** (*dict*) – top border dict.
- **is_delayed** (*bool, optional*) – whether this row is being added in delayed mode (`True`) or not (`False`).

Returns whether `col` is part of a spanned cell (`True`) or not (`False`).

Return type bool

get_cell_style(*element*)

Method to extract the cell style from a cell `element`.

Parameters **element** (*dict, str, list, tuple*) – the cell element to extract the cell style from.

Returns tuple with a copy of `element`, the element `style`, and the `cell_style`.

Return type tuple

is_type(*el*, *type_*)

add_cell(*col*, *element*, *is_delayed*)

Method to add a cell to the current row.

Parameters

- **col** (*int*) – the column index for the cell.

- **element** (*dict, str, list, tuple*) – the cell element to be added.
- **is_delayed** (*bool, optional*) – whether current row is being added in delayed mode (`True`) or not (`False`).

Returns whether `col` is part of a spanned cell (`True`) or not (`False`).

Return type bool

process_text(*element*, *x*, *y*, *width*, *height*, *style*, *delayed*)

Method to add a paragraph to a cell.

Parameters

- **col** (*int*) – the column index of the cell.
- **element** (*dict*) – the paragraph element
- **x** (*Number*) – the x coordinate of the paragraph.
- **y** (*Number*) – the y coordinate of the paragraph.
- **width** (*Number*) – the width of the paragraph.
- **height** (*Number*) – the height of the paragraph.
- **style** (*dict*) – the paragraph style.
- **delayed** (*dict*) – the delayed element to add the current paragraph if it can not be added completely to the current cell.

Returns the height of the paragraph.

Return type float

process_image(*element*, *x*, *y*, *width*, *height*, *delayed*)

Method to add an image to a cell.

Parameters

- **col** (*int*) – the column index of the cell.
- **element** (*dict*) – the image element
- **x** (*Number*) – the x coordinate of the image.
- **y** (*Number*) – the y coordinate of the image.
- **width** (*Number*) – the width of the image.
- **height** (*Number*) – the height of the image.
- **delayed** (*dict*) – the delayed element to add the current image if it can not be added to the current cell.

Returns the height of the image.

Return type float

process_content(*element*, *x*, *y*, *width*, *height*, *style*, *delayed*)

Method to add a content box to a cell.

Parameters

- **col** (*int*) – the column index of the cell.
- **element** (*dict*) – the content box element
- **x** (*Number*) – the x coordinate of the content box.

- **y** (*Number*) – the y coordinate of the content box.
- **width** (*Number*) – the width of the content box.
- **height** (*Number*) – the height of the content box.
- **style** (*dict*) – the content box style.
- **delayed** (*dict*) – the delayed element to add the current content box if it can not be added completely to the current cell.

Returns the height of the content box.

Return type float

process_table(*element*, *x*, *y*, *width*, *height*, *style*, *delayed*)

Method to add a table to a cell.

Parameters

- **col** (*int*) – the column index of the cell.
- **element** (*dict*) – the table element
- **x** (*Number*) – the x coordinate of the table.
- **y** (*Number*) – the y coordinate of the table.
- **width** (*Number*) – the width of the table.
- **height** (*Number*) – the height of the table.
- **style** (*dict*) – the table style.
- **delayed** (*dict*) – the delayed element to add the current table if it can not be added completely to the current cell.

Returns the height of the table.

Return type float

5.3.12 pdfme.text

class pdfme.text.**PDFState**(*style*, *fonts*)

Bases: object

Class that represents the state of a paragraph line part.

The state is a lower level version of the style, and is used by the other paragraph classes to make calculations and yield the paragraph PDF stream.

Parameters

- **style** (*dict*) – the paragraph line part style.
- **fonts** (PDFFonts) – the fonts instance with the information about the fonts already added to the PDF document.

compare(*other*)

Compares this state, with state **other** and returns a PDF stream with the differences between both states.

Parameters **other** (PDFState) – the state to compare.

Returns a PDF stream with the differences between both states.

Return type str

class pdfme.text.**PDFTextLinePart**(*style*, *fonts*, *ids=None*)

Bases: object

This class represents a part of a paragraph line, with its own style.

Parameters

- **style** (*dict*) – the style of this line part.
- **fonts** (PDFFonts) – the fonts instance with the information about the fonts already added to the PDF document.
- **ids** (*list, optional*) – the ids of this part.

pop_word(*index=None*)

Function to delete the last word of this part if index is None, and the word in the position index if it's not None.

Parameters **index** (*int, optional*) – word index.

Returns

if word in index could be deleted, the deleted word is returned, if not None is returned.

Return type str

add_word(*word*)

Function to add a word to this part.

Parameters **word** (*str*) – the word.

current_width(*factor=1*)

Return the width of this part, according to the words added to this part, using factor to calculate this width of the spaces in this part.

Parameters **factor** (*int, float, optional*) – to calculate this width of the spaces in this part.

Returns width of this part.

Return type float

tentative_width(*word*, *factor=1*)

The same as method current_width, but adding the width of word.

Parameters

- **word** (*str*) – the word that could be added to this part.
- **factor** (*int, float, optional*) – to calculate this width of the spaces in this part.

Returns the width of this part + the width of the word passed.

Return type float

get_char_width(*char*)

The width of the character passed.

Parameters **char** (*str*) – the character string.

Returns the width of the character passed.

Return type float

get_word_width(*word*)

The width of the word passed.

Parameters **char** (*str*) – the word string.

Returns the width of the word passed.

Return type float

class pdfme.text.**PDFTextLine**(*fonts*, *max_width=0*, *text_align=None*, *top_margin=0*)

Bases: object

Class that represents a line of a paragraph.

This class has the logic to add paragraph parts, and inside them add their words one by one, until all of the horizontal space of the paragraph has been used. For more information about this mechanism check the method *pdfme.text.PDFTextLine.add_word()*.

Parameters

- **fonts** (PDFFonts) – to extract information about the fonts being used in the paragraph.
- **max_width** (*int, float, optional*) – the maximum horizontal space that this line can use.
- **text_align** (*str, optional*) – 'l' for left (default), 'c' for center, 'r' for right and 'j' for justified text.
- **top_margin** (*Number, optional*) – if not None, this is the top margin of the line, added to the actual line height.

property height

Property that returns the line height, calculated from the vertical space of each part of the line.

Returns the line height.

Return type float

property min_width

Property that returns the width of the line, calculated using the minimum value for attribute factor. This attribute is used to increase or decrease the space character width inside a line to

Returns the line width.

Return type float

property bottom

Property that returns the line bottom, calculated from the vertical space of each part of the line.

Returns the line bottom.

Return type float

get_widths()

This function returns the widths of the line.

Returns

of 2 elements, the width on the words as the first, and the width of the spaces as the second.

Return type tuple

add_line_part(*style=None*, *ids=None*)

Add a new line part to this line.

Parameters

- **style** (*dict, optional*) – the style of the new part.
- **ids** (*list, optional*) – the ids of the new part.

Returns The new line part that was added.

Return type *PDFTextLinePart*

add_accumulated()

Function to add the parts accumulated in the auxiliar line (`next_line` attribute) to this line.

add_word(*word*)

Function to add a word to this line.

Parameters **word** (*str*) – The word to be added.

Returns

containing a `status` key, with one of the following values:

- `'added'`: The word passed was added to the auxiliar line, or if the word is a space the accumulated words in the auxiliar line, to the current line.
- `'ignored'`: The word passed (a space) was ignored.
- `'preadded'`: The word passed was added to the auxiliar line.
- `'finished'`: The word didn't fit in the current line, and this means this line is full. Because of this, a new line is created to put this word, and this new line is returned in the key `'new_line'`.

Return type dict

class pdfme.text.**PDFTextBase**(*content*, *width*, *height*, *x=0*, *y=0*, *fonts=None*, *text_align=None*, *line_height=None*, *indent=0*, *list_text=None*, *list_indent=None*, *list_style=None*, *pdf=None*)

Bases: `object`

Class that represents a rich text paragraph to be added to a *pdfme.pdf.PDF* instance.

You should use *pdfme.text.PDFText* instead of this class, because it has more functionalities.

To create the data needed to add this paragraph to the PDF document, you have to call the method *pdfme.text.PDFTextBase.run()*, which will try to add all of the dict parts in `content` argument list (or tuple) to the rectangle defined by args `x`, `y`, `width` and `height`.

Each part represents a part of the paragraph with a different style or with a `var` or a specific `id`.

The parts are added to this rectangle, until they are all inside of it, or until all of the vertical space is used and the rest of the parts can not be added. In these two cases method `run` finishes, and the property `finished` will be True if all the parts were added, and False if the vertical space ran out. If `finished` is False, you can set a new rectangle (on a new page for example) and use method `run` again (passing the parameters of the new rectangle) to add the remaining parts that couldn't be added in the last rectangle. You can keep doing this until all of the parts are added and therefore property `finished` is True.

By using method `run` the paragraph is not really added to the PDF object. After calling `run`, the property `result` will be available with the information needed to be added to the PDF, at least the parts that fitted inside the rectangle. You have to use the property `result` to add the paragraph to the PDF object before using method

`run` again (in case `finished` is False), because it will be redefined for the next rectangle after calling `run` again. You can check the `text` method in PDF module to see how this process is done.

The other args not defined here, are explained in *pdfme.text.PDFText*.

Parameters **content** (*str, list, tuple*) – If this is a string, it will become the following:

```
[{'style': <DEFAULT_STYLE>, 'text': <STRING>}]
```

If this is a list or a tuple, its elements should be dicts with the following keys:

- `'text'`: this is the text that will be displayed with the style defined in `style` key.
- `'style'`: this is a style dict like the one described in *pdfme.text.PDFText*.
- `'ids'`: see *pdfme.text.PDFText* definition.
- `'var'`: see *pdfme.text.PDFText* definition.

Raises **TypeError** – if `content` is not a str, list or tuple.

property stream

Property that returns the PDF stream generated by the method `run`, with all of the graphics and the text, ready to be added to a PDF page stream.

Returns the stream.

Return type str

property result

Property that returns a dict with the result of calling method `run`, and can be passed to method `pdfme.pdf.PDF._add_text()`, to add this paragraph to that PDF document's page. Check method `_add_parts` from *pdfme.pdf.PDF* to see how a dict like the one returned by this method (a paragraph part) is added to a PDF instance.

The dict returned will have the following keys:

- `x` the x coordinate.
- `y` the y coordinate.
- `width` of the paragraph.
- `height` of the paragraph.
- `text_stream` a string with the paragraphs PDF text stream.
- `graphics_stream` a string with the paragraphs PDF graphics stream.
- `used_fonts` a set with tuples of 2 elements, first element the font family, and second element the font mode.
- `ids` a dict with every id extracted from the paragraph.

Returns like the one described.

Return type dict

get_state()

Method to get the current state of this paragraph. This can be used later in method `pdfme.text.PDFText.set_state()` to restore this state in this paragraph (like a checkpoint in a videogame).

Returns a dict with the state of this paragraph.

Return type dict

`set_state`(*last_part=None*, *last_word=None*)

Function to update the state of the paragraph

The arguments of this method define the current state of this paragraph, and with this method you can change that state.

Parameters

- **last_part** (*int*) – this is the index of the part that was being processed the last time method `run` was called.
- **last_word** (*int*) – this is the index of the word of the last part that was added the last time method `run` was called.

`setup`(*x=None*, *y=None*, *width=None*, *height=None*)

Function to change any or all of the parameters of the rectangle of the content.

Parameters

- **x** (*int, float, optional*) – The x coordinate of the left of the rectangle.
- **y** (*int, float, optional*) – The y coordinate of the top of the rectangle.
- **width** (*int, float, optional*) – The width of the rectangle where the contents will be arranged.
- **height** (*int, float, optional*) – The height of the rectangle where the contents will be arranged.
- **last_part** (*int, optional*) – If not None, this is the index of the part that was being processed the last time method `run` was called.
- **last_word** (*int, optional*) – If not None, this is the index of the word of the last part that was added the last time method `run` was called.

`init`()

Function to reset all of the instance properties that have to be resetted before running the arranging process in a new rectangle.

This function is called by method `run`.

`run`(*x=None*, *y=None*, *width=None*, *height=None*)

Function to create the data needed to add this paragraph to the PDF document.

This function will try to add all of the dict parts in `content` argument list (or tuple) to this paragraph rectangle. Check this class documentation for more information about this method.

This function args are the same as *pdfme.text.PDFTextBase.setup()*.

Returns The dict from the property `result`.

Return type dict

`add_part`(*part*, *part_index*)

Function used by methodm `run` to add one paragraph part at a time.

Parameters

- **part** (*dict*) – part to be added.
- **part_index** (*int*) – index of the part to be added.

Returns

whether it was able to add all of the parts (True) or the vertical space ran out.

Return type bool

`add_current_line`(*is_last=False*)

Function to add the current line to the list of already added lines.

Parameters **is_last** (*bool, optional*) – whether this is the last line of this paragraph (True) or not (False).

Returns whether this line was successfully added (True) or not (False)

Return type bool

`setup_list`()

This function is called when the first part of the paragraph is going to be added, and if this is a list paragraph, i.e. a paragraph with a something on its left (a bullet, a number, etc), this function will setup everything needed to display the text of the list paragraph, and will adjust its width to make space for the list text.

Raises **TypeError** – if list_style or list_indent passed to this instance are not a dict and an number respectively.

`add_line_to_stream`(*line*, *is_last=False*)

Function to add a PDFTextLine representing a paragraph line to the already added lines stream.

Parameters

- **line** (PDFTextLine) – The line to be added to the stream.
- **is_last** (*bool, optional*) – whether this is the last line of this paragraph (True) or not (False).

`clean_words`(*words*)

This function joins a list of words (spaces included) and makes the resulting string compatible with a PDF string.

Parameters **words** (*list*) – a list of strings, where each string is a word.

Returns A string with all of the words passed.

Return type str

`output_text`(*part*, *text*, *factor=1*)

Function that creates a piece of PDF stream (only the text), from the PDFTextLinePart and the `text` arguments.

Parameters

- **part** (PDFTextLinePart) – the part to be transformed into a string representing a PDF stream piece.
- **text** (*[type]*) – the text to be transformed into a string representing a PDF stream piece.
- **factor** (*Number, optional*) – factor of the line needed to create center, right and justified aligned paragraphs.

Returns representing the PDF stream

Return type str

`output_graphics`(*part*, *x*, *y*, *part_width*)

Function that creates a piece of PDF stream (only the graphics), from the PDFTextLinePart argument.

Parameters

- **part** (PDFTextLinePart) – the part to be transformed into a string representing a PDF stream piece.

- **x** (*int, float*) – the x origin coordinate of the graphics being added.
- **y** (*int, float*) – the y origin coordinate of the graphics being added.
- **width** (*int, float*) – the width of the part being added

Returns representing the PDF stream

Return type str

class pdfme.text.**PDFText**(*content*, *width*, *height*, *x=0*, *y=0*, *fonts=None*, *text_align=None*, *line_height=None*, *indent=0*, *list_text=None*, *list_indent=None*, *list_style=None*, *pdf=None*)

Bases: *pdfme.text.PDFTextBase*

Class that represents a rich text paragraph to be added to a *pdfme.pdf.PDF* instance.

`content` argument should be a dict, with a key starting with a dot, like `'.b;s:10;c:1;u'` for example (keep reading to learn more about the format of this key), which we are going to refer to as the "the dot key" from here on. The value for the dot key is a list/tuple containing strings or more `content` dicts like the one we are describing here (you can have nested `content` dicts or what we call a paragraph part), but for simplicity, you can pass a string (for non-rich text) or a tuple/list with strings and more paragraph parts:

- If `content` argument is a string, it will become the following:

```
{ '.': [ <STRING>, ] }
```

- If `content` argument is a list/tuple, it will become the following:

```
{ '.': <LIST_OR_TUPLE> }
```

This is an example of a `content` argument:

```
{
    ".b;u;i;c:1;bg:0.5;f:Courier": [
        "First part of the paragraph ",
        {
            ".b:0;u:0;i:0;c:0;bg:": [
                "and here the second, nested inside the root paragraph,",
            ]
        },
        "and yet one more part before a ",
        {".c:blue;u:1": "url link", "uri": "https://some.url.com"}
    ]
}
```

This class is a subclass of *pdfme.text.PDFTextBase* and adds the logic to let the user of this class pass content in a nested cascading "jsonish" format (like HTML), i.e. if you pass a dict to `content`, and this dict has a `style` key, all of its children will inherit this style and will be able to overwrite some or all of the style parameters coming from it. The children will be able to pass their own style parameters to their children too, and so on.

Additional to the dot key, paragraph parts can have the following keys:

- `'label'`: this is a string with a unique name (there should be only one label with this name in the whole document) representing a destination that can be referenced in other parts of the document. This is suited for titles, figures, tables, etc.
- `'ref'`: this is a string with the name of a label, that will become a link to the position of the label referenced.
- `'uri'`: this is a string with a reference to a web resource, that will turn this text part in a link to that web page.

- `'outline'`: an outline is a label that is shown in the outlines panel of the PDF reader. This outlines show the structure of the document. This attribute is a dict with the following optional keys:
  - `level`: an optional int with the level of this outline in the outlines tree. The default value is 1.
  - `text`: an optional string to be shown in the outlines panel for this outline. The default value is the contents of this part.
- `'ids'`: when method `run` is called, dict attr `result` is available with information to add the paragraph to the PDF, and within that information you'll find a key `ids`, a dict with the position and size of the rectangle for each of the ids you include in this argument. This way you can "tag" a part of a paragraph, call `run`, and get the position of it afterwards.
- `'var'`: this is a string with the name of a global variable, previously set in the containing *pdfme.pdf.PDF* instance, by adding a new key to its dict attribute `context`. This way you can reuse a repetitive string throughout the PDF document.

Style of the paragraph dicts can be defined in the dot key itself (a string with a semi-colon separeted list of the attributes, explained in *pdfme.utils.parse_style_str()*) or in a `style` dict too. The attributes for a paragraph style are the following:

- `'b'` (bool) to make text inside this part bold. Default is False.
- `'i'` (bool) to make text inside this part cursive (italics, oblique). Default is False.
- `'s'` (int, float) to set the size of the text inside this part. Default is 11.
- `'f'` (str) to set the font family of the text inside this part. Default is `'Helvetica'`.
- `'u'` (bool) to make the text inside this part underlined. Default is False.
- `'c'` (int, float, list, tuple, str) to set the color of the text inside this part. See *pdfme.color.parse_color()* for information about this attribute. Default is black.
- `'bg'` (int, float, list, tuple, str) to set the background color of the text inside this part. See *pdfme.color.parse_color()* for information about this attribute. Default is None.
- `'r'` (int, float) to set the baseline of the text, relative to the normal baseline. This is a fraction of the current size of the text, i.e. it will move the baseline the text size times this number in points, upwards if positive, and downwards if negative. Default is 0.

One more example of a `content` argument with a `style` dict, and additional keys:

```
{
    '.': ['text to be displayed'],
    'style': {
        'b': True,
        'i': True,
        'u': True,
        's': 10.2,
        'f': 'Courier',
        'c': 0.9,
        'bg': 'red',
        'r': 0.5
    },
    'label': 'a_important_paragraph',
    'uri': 'https://github.com/aFelipeSP/pdfme'
}
```

With arguments `list_text`, `list_indent` and `list_style` you can turn a paragraph into a list paragraph, one that has a bullet or a number at the left of the paragraph, with an additional indentation. With this you can build a bulleted or numbered list of paragraphs.

Parameters

- **content** (*str, list, tuple, dict*) – the one just described.
- **width** (*int, float*) – The width of the paragraph.
- **height** (*int, float*) – The height of the paragraph.
- **x** (*int, float, optional*) – The x coordinate of the paragraph.
- **y** (*int, float, optional*) – The y coordinate of the paragraph
- **fonts** (PDFFonts, *optional*) – To extract information about the fonts being used in the paragraph.
- **text_align** (*str, optional*) – `'l'` for left (default), `'c'` for center, `'r'` for right and `'j'` for justified text.
- **line_height** (*int, float, optional*) – How much space between the lines of the paragraph. This is a fraction of each line's height, so the real distance between lines can vary depending on the text size of each part of the paragraph.
- **indent** (*int, float, optional*) – The indentation of the first line of the paragraph.
- **list_text** (*str, optional*) – Needed if you want to turn this paragraph into a list paragraph. This text will be displayed before the paragraph and will be aligned with the first line.
- **list_indent** (*int, float, optional*) – Needed if you want to turn this paragraph into a list paragraph. The space between the start of the left side of the rectangle and the left side of the paragraph itself. If omitted, this space will be the width of the `list_text`.
- **list_style** (*dict, optional*) – Needed if you want to turn this paragraph into a list paragraph. The style of `list_text`. If omitted, the style of the first part of the first line will be used.
- **pdf** (PDF, *optional*) – To grab global information of the PDF being used.

5.3.13 pdfme.utils

pdfme.utils.**subs**(*string*, **args*, ***kwargs*)

Function to take `string`, format it using `args` and `kwargs` and encode it into bytes.

Parameters **string** (*str*) – string to be transformed.

Returns the resulting bytes.

Return type bytes

pdfme.utils.**process_style**(*style*, *pdf=None*)

Function to use a named style from the PDF instance passed, if `style` is a string or `style` itself if this is a dict.

Parameters

- **style** (*str, dict*) – a style name (str) or a style dict.
- **pdf** (PDF, *optional*) – the PDF to extract the named style from.

Returns a style dict.

Return type dict

pdfme.utils.**get_page_size**(*size*)

Function to get tuple with the width and height of a page, from the value in `size`.

If `size` is a str, it should be the name of a page size: `a5`, `a4`, `a3`, `b5`, `b4`, `jis-b5`, `jis-b4`, `letter`, `legal` and `ledger`.

If `size` is a int, the page will be a square of size `(int, int)`.

If `size` is a list or tuple, it will be converted to a tuple.

Parameters **size** (*int, float, str, iterable*) – the page size.

Returns tuple with the page width and height.

Return type tuple

pdfme.utils.**parse_margin**(*margin*)

Function to transform `margin` into a dict containing keys `top`, `left`, `bottom` and `right` with the margins.

If `margin` is a dict, it is returned as it is.

If `margin` is a string, it will be splitted using commas or spaces, and each substring will be converted into a number, and after this, the list obtained will have the same treatment of an iterable.

If `margin` is an iterable of 1 element, its value will be the margin for the four sides. If it has 2 elements, the first one will be the `top` and `bottom` margin, and the second one will be the `left` and `right` margin. If it has 3 elements, these will be the `top`, `right` and `bottom` margins, and the `left` margin will be the second number (the same as `right`). If it has 4 elements, they will be the `top`, `right`, `bottom` and `left` margins respectively.

Parameters **margin** (*str, int, float, tuple, list, dict*) – the margin element.

Returns dict containing keys `top`, `left`, `bottom` and `right` with the margins.

Return type dict

pdfme.utils.**parse_style_str**(*style_str*, *fonts*)

Function to parse a style string into a style dict.

It parses a string with a semi-colon separeted list of the style attributes you want to apply (for a list of the attributes you can use in this string see *pdfme.text.PDFText*). For the ones that are of type bool, you just have to include the name and it will mean they are `True`, and for the rest you need to include the name, a colon, and the value of the attribute. In case the value is a color, it can be any of the possible string inputs to function *pdfme.color.parse_color()*. Empty values mean `None`, and `"1" == True` and `"0" == False` for bool attributes.

This is an example of a valid style string:

```
".b;s:10;c:1;u:0;bg:"
```

Parameters

- **style_str** (*str*) – The string representing the text style.
- **fonts** (PDFFonts) – If a font family is included, this is needed to check if it is among the fonts already added to the PDFFonts instance passed.

Raises **ValueError** – If the string format is not valid.

Returns A style dict like the one described in *pdfme.text.PDFText*.

Return type dict

pdfme.utils.**create_graphics**(*graphics*)

Function to transform a list of graphics dicts (with lines and fill rectangles) into a PDF stream, ready to be added to a PDF page stream.

Parameters **graphics** (*list*) – list of graphics dicts.

Returns a PDF stream containing the passed graphics.

Return type str

pdfme.utils.**to_roman**(*n*)

Function to transform `n` integer into a string with its corresponding Roman representation.

Parameters **n** (*int*) – the number to be transformed.

Returns the Roman representation of the integer passed.

Return type str

pdfme.utils.**get_paragraph_stream**(*x*, *y*, *text_stream*, *graphics_stream*)

Function to create a paragraph stream, in position `x` and `y`, using stream information in `text_stream` and `graphics_stream`.

Parameters

- **x** (*int, float*) – the x coordinate of the paragraph.
- **y** (*int, float*) – the y coordinate of the paragraph.
- **text_stream** (*str*) – the text stream of the paragraph.
- **graphics_stream** (*str*) – the graphics stream of the paragraph.

Returns the whole stream of the paragraph.

Return type str

pdfme.utils.**copy**(*obj*)

Function to copy objects like the ones used in this project: dicts, lists, PDFText, PDFTable, PDFContent, etc.

Parameters **obj** (*Any*) – the object to be copied.

Returns the copy of the object passed as argument.

Return type Any

class pdfme.utils.**MuiltiRange**

Bases: `object`

add(**range_args*)

pdfme.utils.**parse_range_string**(*range_str*)

Function to convert a string of comma-separated integers and integer ranges into a set of all the integers included in those.

Parameters **range_str** (*str*) – comma-separated list of integers and integer ranges.

Returns a set of integers.

Return type *MuiltiRange*

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

Symbols

A

B

C

E

F

G

H

I

L

M

N

O

P

R

S

T

U

W

Y