N2

metin kurtusu

kubikanber

heryer

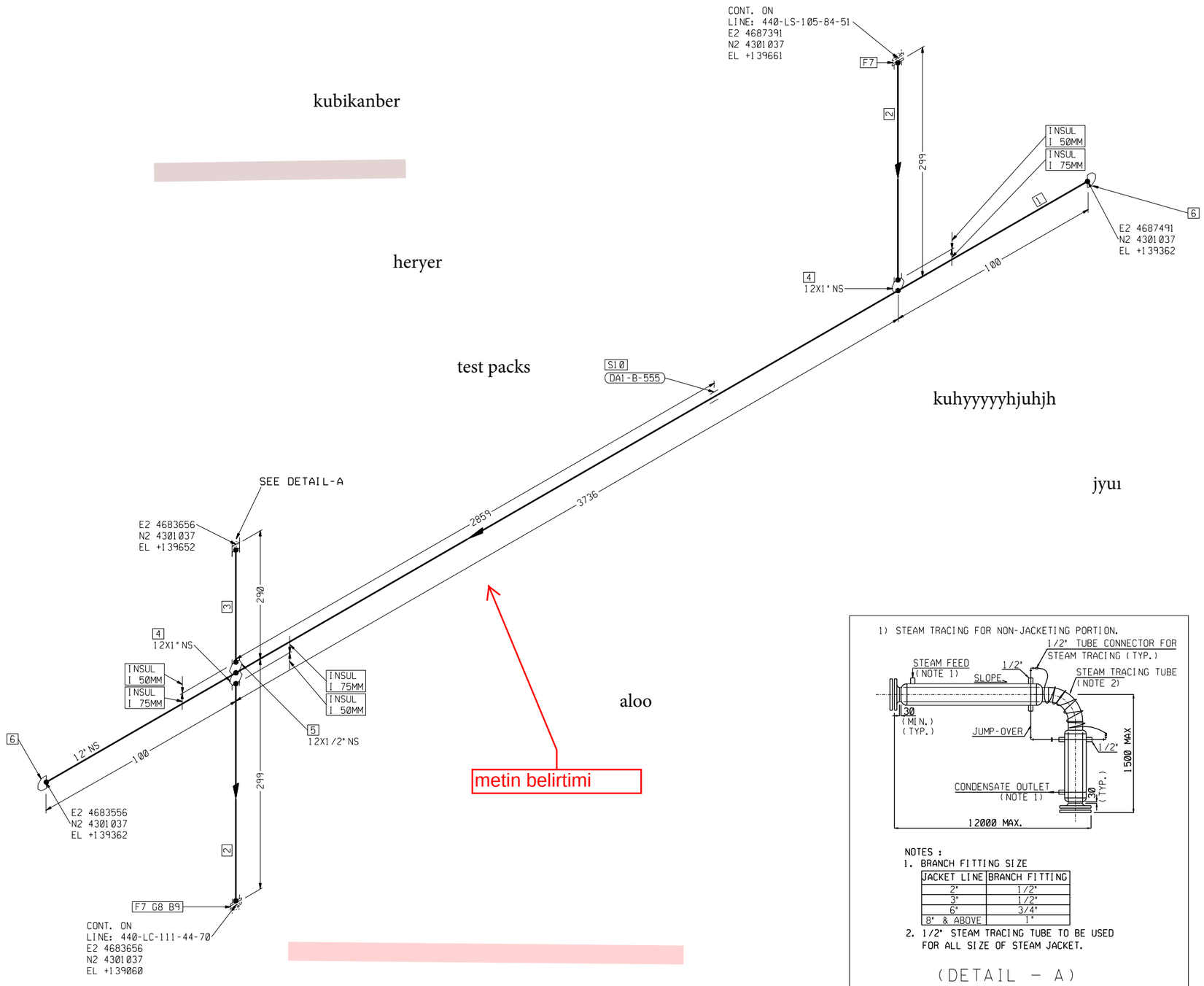
test packs

kuhyyyyyhjuhjh

jyuı

aloo

metin belirtimi



NOTES :
1. BRANCH FITTING SIZE

JACKET LINE	BRANCH FITTING
2"	1/2"
3"	1/2"
6"	3/4"
8" & ABOVE	1"

2. 1/2" STEAM TRACING TUBE TO BE USED FOR ALL SIZE OF STEAM JACKET.

(DETAIL - A)

JACKETED OUTER PIPE
REF. NO.:-440-P-008-14-52

BILL OF MATERIALS

PT NO	COMPONENT DESCRIPTION	N.S. (INS)	ITEM CODE	QTY
	PIPE			
1	PIPE A106-B SEAMLESS - - - - BE 12 IN. S-STD	12	15681	3936 MM
2	PIPE A106-B SEAMLESS - - - - PE 1 IN. S-80	1	11196254	200 MM
3	PIPE A106-B SEAMLESS - - - - PE .5 IN. S-80	1/2	11196252	100 MM
	FITTINGS			
4	SOCKOLET A105 - - - 3000 SW 12 IN. 1 IN.	12X1	191071	2
5	SOCKOLET A105 - - - 3000 SW 12 IN. .5 IN.	12X1/2	191069	1
6	CAP A234-WPB - - BW - 12 IN. S-STD	12	18120	2
	FLANGES			
7	SOCKETWELD FLANGE A105 - - 300 RF - 1 IN. S-80	1	11197972	2
	GASKETS			
8	SPIRAL WOUND GASKET 316/GRAPH. CS-CR/316-IR 300 BETW.FLG RF 4.5 MM 1 IN.	1	11210933	1

PT NO	COMPONENT DESCRIPTION	N.S. (INS)	ITEM CODE	QTY
	BOLTS			
9	STD BOLT 2HY HX NUT A193 B7/2H D: .625 IN. L: 3.5 IN., 3.5" BOLT LENGTH	5/8	11520665	4
	SUPPORTS			
10	DA1-B-555	12	PSUPPORT	1

REV.	DATE	DESCRIPTION	BY	STRESS SUPPORT	CHD.	APPR.
0	18-MAY-16	APPROVED FOR CONSTRUCTION	SNS	CSP	USS	YJK

REVISIONS

235	7.6	12			X
DESIGN TEMP	DESIGN PRESS	TEST PRESS	YES	NO	AS PER SPECIFICATION 000-L-EE-0020003
			PWHT		

NOTE:
IN ALL ISOMETRICS BELOW 2 IN ND, THE DIMENSIONS SHOULD BE ADJUSTED AT SITE.
ALL ISOMETRICS 2 IN ND AND BELOW, HAVE TO BE SUPPORTED AT SITE, UNLESS OTHERWISE SHOWN IN THE ISOMETRICS.

CONTRACT No: STAR-EP-01
JOB No: 2245
REQ. No:
PLANT LOCATION: İZMİR-TURKEY

STAR RAFINERI, A.S
AEGEAN REFINERY PROJECT (ARP)

LINE NUMBER
12"-LS440-008-14-AA28S-I

PLANT:	SECTION:	INDEX:	ISOMETRIC DWG:	TRAIN No:	REV.
440	000	L	LS-008-14	58	0

CAD FILE: 440-LS-008-14-58

ALL COORDINATES AND ELEVATIONS ARE REFERED TO CENTER LINE (℄) OF PIPES
MODEL NUMBER: PJ420R0102
EXTRACTION DATE: 16-MAY-16
FILENAME: 440-LS-008-14-58.dgn

sayfa 01 metin 01

Extracting Text & Images from PDF Files

August 04, 2010

sırada değişimi

Update: *January 29, 2012*
I've corrected this code to work with the current version of pdfminer and it's now available as a github repo: https://github.com/dpapathanasiou/pdfminer-layout-scanner

enson bu

PDFMiner is a pdf parsing library written in Python by Yusuke Shinyama.

In addition to the pdf2txt.py and dumppdf.py command line tools, there is a way of analyzing the content tree of each page.

Since that's exactly the kind of programmatic parsing I wanted to use PDFMiner for, this is a more complete example, which continues where the default documentation stops.

This example is still a work-in-progress, with room for improvement.

In the next few sections, I describe how I built up each function, resolving problems I encountered along the way. The impatient can just get the code here instead.

Basic Framework

Here are the python imports we need for PDFMiner:

```
from pdfminer.pdfparser import PDFParser, PDFDocument, PDFNoOutlines
from pdfminer.pdfinterp import PDFResourceManager, PDFPageInterpreter
from pdfminer.converter import PDFPageAggregator
from pdfminer.layout import LAParams, LTTextBox, LTTextLine, LTFigure, LTImage
```

Since PDFMiner requires a series of initializations for each pdf file, I've started with this wrapper (Lisp macro style) function to take care of the basic preliminary actions (file IO, PDFMminer object creation and connection, etc.).

```
def with_pdf (pdf_doc, pdf_pwd, fn, *args):
    """Open the pdf document, and apply the function, returning the results"""
    result = None
    try:
        # open the pdf file
        fp = open(pdf_doc, 'rb')
        # create a parser object associated with the file object
        parser = PDFParser(fp)
        # create a PDFDocument object that stores the document structure
        doc = PDFDocument()
        # connect the parser and document objects
        parser.set_document(doc)
        doc.set_parser(parser)
        # supply the password for initialization
        doc.initialize(pdf_pwd)

        if doc.is_extractable:
            # apply the function and return the result
            result = fn(doc, *args)

        # close the pdf file
        fp.close()
    except IOError:
        # the file doesn't exist or similar problem
        pass
    return result
```

The first two parameters are the name of the pdf file, and its password. The third parameter, fn, is a higher-order function which takes the instance of the pdfminer.pdfparser.PDFDocument created, and applies whatever action we want (get the table of contents, walk through the pdf page by page, etc.)

*The last part of the signature, *args, is an optional list of parameters that can be passed to the high-order function as needed (I could have gone with keyword arguments here instead, but a simple list is enough for these examples).*

As a warm-up, here's an example of how to use the with_pdf() function to fetch the table of contents from a pdf file:

```
def _parse_toc (doc):
    """With an open PDFDocument object, get the table of contents (toc) data
    [this is a higher-order function to be passed to with_pdf()]"""
    toc = []
    try:
        outlines = doc.get_outlines()
        for (level,title,dest,a,se) in outlines:
            toc.append( (level, title) )
    except PDFNoOutlines:
        pass
    return toc
```

The _parse_toc() function is the higher-order function which gets passed to with_pdf() as the fn parameter. It expects a single parameter, doc, which is the the instance of the pdfminer.pdfparser.PDFDocument created within with_pdf() itself (note that if with_pdf() couldn't find the file, then _parse_toc() doesn't get called).

With all the PDFMiner overhead and initialization done by with_pdf(), _parse_toc() can just focus on collecting the table of content data and returning them as a list. The get_outlines() can raise a "PDFNoOutlines" error, so I catch it as an exception, and simply return an empty list in that case.

All that's left to do is define the function that invokes _parse_toc() for a specific pdf file; this is also the function that any external users of this module would use to get the table of contents list. Note that the pdf password defaults to an empty string (which is what PDFMiner will use for documents that aren't password-protected), but that can be overriden as needed.

```
def get_toc (pdf_doc, pdf_pwd=''):
    """Return the table of contents (toc), if any, for this pdf file"""
    return with_pdf(pdf_doc, pdf_pwd, _parse_toc)
```

Page Parsing

Next, onto layout analysis. Using the with_pdf() wrapper, we can reproduce the example in the documentation with this higher-order function:

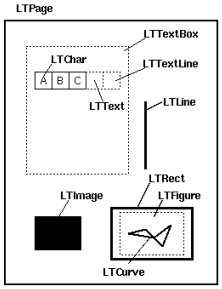
```
def _parse_pages (doc):
    """With an open PDFDocument object, get the pages and parse each one
    [this is a higher-order function to be passed to with_pdf()]"""
    rsrcmgr = PDFResourceManager()
    laparams = LAParams()
    device = PDFPageAggregator(rsrcmgr, laparams=laparams)
    interpreter = PDFPageInterpreter(rsrcmgr, device)

    for page in doc.get_pages():
        interpreter.process_page(page)
        # receive the LTPage object for this page
        layout = device.get_result()
        # layout is an LTPage object which may contain child objects like LTTextBox, LTFigure, LTImage, etc.
```

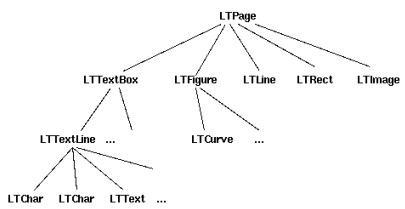
And this external function, which defines the specific pdf file to analyze:

```
def get_pages (pdf_doc, pdf_pwd=''):
    """Process each of the pages in this pdf file"""
    with_pdf(pdf_doc, pdf_pwd, _parse_pages)
```

So far, this code doesn't do anything exciting: it just loads each page into a pdfminer.layout.LTPage object, closes the pdf file, and exits.

Within each pdfminer.layout.LTPage instance, though, is an objs attribute, which defines the tree of pdfminer.layout.LT child objects as in the documentation:*





In this example, I'm going to collect all the text from each page in a top-down, left-to-right sequence, merging any multiple columns into a single stream of consecutive text.

The results are not always perfect, but I'm using a fuzzy logic based on physical position and column width, which is very good in most cases.

I'm also going to save any images found to a separate folder, and mark their position in the text with <img /> tags.

Right now, I'm only able to extract jpeg images, whereas xpdf's pdfimages tool is capable of getting to non-jpeg images and saving them as ppm format.

I'm not sure if the problem is within PDFMiner or how I'm using it, but since someone else asked the same question in the PDFMiner mailing list, I suspect it's the former.

This requires a few updates to the _parse_pages() function, as follows:

```
def _parse_pages (doc, images_folder):
    """With an open PDFDocument object, get the pages, parse each one, and return the entire text
    [this is a higher-order function to be passed to with_pdf()]"""
    rsrcmgr = PDFResourceManager()
    laparams = LAParams()
    device = PDFPageAggregator(rsrcmgr, laparams=laparams)
    interpreter = PDFPageInterpreter(rsrcmgr, device)

    text_content = [] # a list of strings, each representing text collected from each page of the doc
    for i, page in enumerate(doc.get_pages()):
        interpreter.process_page(page)
        # receive the LTPage object for this page
        layout = device.get_result()
        # layout is an LTPage object which may contain child objects like LTTextBox, LTFigure, LTImage, etc.
        text_content.append(parse_lt_objs(layout.objs, (i+1), images_folder))

    return text_content
```

and the updated get_pages() function becomes:

```
def get_pages (pdf_doc, pdf_pwd='', images_folder='/tmp'):
    """Process each of the pages in this pdf file and print the entire text to stdout"""
    print '\n\n'.join(with_pdf(pdf_doc, pdf_pwd, _parse_pages, *tuple([images_folder])))
```

*New in both functional signatures is images_folder, which is a parameter that refers to the place on the local filesystem where any extracted images will be be saved (this is also an example of why defining with_pdf() with an optional *args list comes in handy).*

Aggregating Text

Within the _parse_pages() function, text_content is a new variable of type list, which collects the text of each page, and I've added an enumeration structure around doc.get_pages(), to keep track of which page we're accessing at any given time. This is useful for saving images correctly, since some pdf files use the same image name in multiple places to refer to different images (this creates problems for dumppdf.py's -i switch, for example).

The new critical line in _parse_pages() is this one:

```
text_content.append(parse_lt_objs(layout.objs, (i+1), images_folder))
```

Since the tree of page objects is recursive in nature (e.g., a pdfminer.layout.LTFigure object may have multiple child objects), it's better to handle the actual text parsing and image collection in a separate function. That function, parse_lt_objs(), looks like this:

```
def parse_lt_objs (lt_objs, page_number, images_folder, text=[]):
    """Iterate through the list of LT* objects and capture the text or image data contained in each"""
    text_content = []

    for lt_obj in lt_objs:
        if isinstance(lt_obj, LTTextBox) or isinstance(lt_obj, LTTextLine):
            # text
            text_content.append(lt_obj.get_text())
        elif isinstance(lt_obj, LTImage):
            # an image, so save it to the designated folder, and note it's place in the text
            saved_file = save_image(lt_obj, page_number, images_folder)
            if saved_file:
                # use html style <img /> tag to mark the position of the image within the text
                text_content.append('<img src="'+os.path.join(images_folder, saved_file)+'" />')
            else:
                print >> sys.stderr, "Error saving image on page", page_number, lt_obj.__repr__
        elif isinstance(lt_obj, LTFigure):
            # LTFigure objects are containers for other LT* objects, so recurse through the children
            text_content.append(parse_lt_objs(lt_obj.objs, page_number, images_folder, text_content))

    return '\n'.join(text_content)
```

In this example, I'm concerned with just four objects which may appear within a pdfminer.layout.LTPage object:

1. *LTTextBox and LLTextLine (which, because the text extraction is exactly the same, I treat as one case)*
2. *LTImage (which we'll try to save on to the local filesystem in the designated folder)*
3. *LTFigure (which we'll treat as a simple container for other objects, hence the recursive call in that case)*

For the simple text and image extraction I'm doing here, this is enough. There is room for improvement, though, since I'm ignoring several types of pdfminer.layout.LT objects which do appear in pdf pages.*

If you try to run get_pages() now, you might get this error, in the text_content.append(lt_obj.get_text()) line (it will depend on the content of the pdf file you're trying to parse, as well as how your instance of Python is configured, and whether or not you installed PDFMiner with cmap for CJK languages).

```
UnicodeEncodeError: 'ascii' codec can't encode character u'\u2014' in position 61: ordinal not in range(128)
```

As Eliot explains, "This error occurs when you pass a Unicode string containing non-English characters (Unicode characters beyond 128) to something that expects an ASCII bytestring. The default encoding for a Python bytestring is ASCII."

This function, which I wrote after reading this article, solves the problem:

```
def to_bytestring (s, enc='utf-8'):
    """Convert the given unicode string to a bytestring, using the standard encoding,
    unless it's already a bytestring"""
    if s:
        if isinstance(s, str):
            return s
        else:
            return s.encode(enc)
```

So the updated version of parse_lt_objs() becomes:

```
def parse_lt_objs (lt_objs, page_number, images_folder, text=[]):
    """Iterate through the list of LT* objects and capture the text or image data contained in each"""
    text_content = []

    for lt_obj in lt_objs:
        if isinstance(lt_obj, LTTextBox) or isinstance(lt_obj, LTTextLine):
            # text
            text_content.append(lt_obj.get_text())
        elif isinstance(lt_obj, LTImage):
            # an image, so save it to the designated folder, and note it's place in the text
            saved_file = save_image(lt_obj, page_number, images_folder)
            if saved_file:
                # use html style <img /> tag to mark the position of the image within the text
                text_content.append('<img src="'+os.path.join(images_folder, saved_file)+'" />')
            else:
                print >> sys.stderr, "Error saving image on page", page_number, lt_obj.__repr__
        elif isinstance(lt_obj, LTFigure):
            # LTFigure objects are containers for other LT* objects, so recurse through the children
            text_content.append(parse_lt_objs(lt_obj.objs, page_number, images_folder, text_content))

    return '\n'.join(text_content)
```

Running this version gives reasonable results on pdf files where the text is single-column, and without many sidebars, abstracts, summary quotes, or other fancy typesetting layouts.

It really breaks down, though, in the case of multi-column pages: the resulting text_content jumps from one paragraph to the next, in no coherent order.

PDFMiner does provide two grouping functions, group_textbox_lr_tb and group_textbox_tb_rl [lr=left-to-right, tb=top-to-bottom], but they do the grouping literally, without considering the likelihood that the content of one textbox logically belongs after another's.

Fortunately, though, each object also provides a bbox (bounding box) attribute, which is a four-part tuple of the object's page position: (x0, y0, x1, y1).

Using the bbox data, we can group the text according to its position and width, making it more likely the columns we join together this way represent the correct logical flow of the text.

To aggregate the text this way, I added the following Python dictionary variable to the parse_lt_objs() code, just before iterating through the list of lt_objs: page_text={}.

The key for each entry is a tuple of the bbox's (x0, x1) points, and the corresponding value is a list of text strings found within that bbox. The x0 value tells me the left offset for a given piece of text and the x1 value tells me how wide it is.

So by grouping text which starts at the same horizontal plane and has the same width, I can aggregate all paragraphs belonging to the same column, regardless of their vertical position or length.

Conceptually, each entry in the page_text dictionary represents all the text associated with each physical column.

When I tried this the first time, I was surprised (though in retrospect, I shouldn't have been, since nothing about parsing pdfs is neat or clean), that two textboxes which look perfectly aligned visually have slightly different x0 and x1 values (at least according to PDFMiner).

For example, one paragraph may have x0 and x1 values of 28.16 and 153.32 respectively, and the paragraph right underneath it had an x0 value of 29.04 and an x1 value of 152.09.

To get around this, I wrote the following update function, which assigns key tuples based on how close an (x0, x1) pair lies within an existing entry's key. The 20 percent value was arrived at by trial-and-error, and seems to be acceptable for most pdf files I tried.