## Data extraction from a PDF table with semi-structured layout

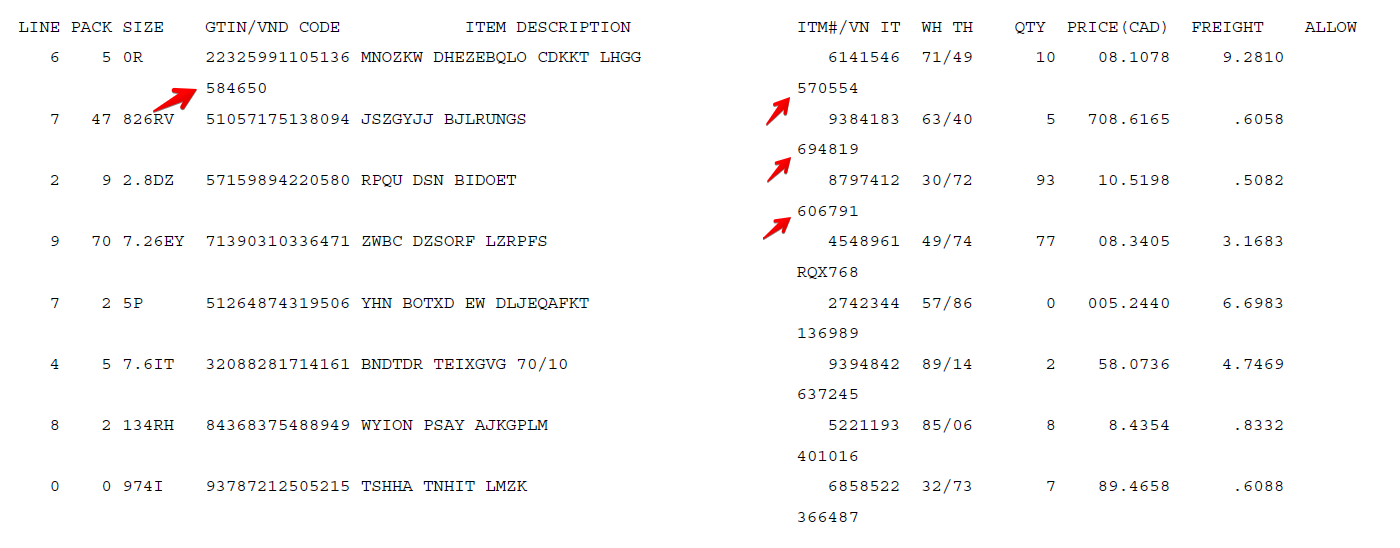
### What is semi-structured data?

In today's work environment PDF documents are widely used for exchanging business information, internally as well as with trading partners. Naturally, you've seen quite a lot of PDFs in the form of invoices, purchase orders, shipping notes, price-lists etc. Despite serving as a digital replacement of paper PDF documents present a challenge for automated manipulation with data they store. It is as accessible as data written on a piece of paper since some PDFs are designed to transfer information to us, humans, but not computers. Such PDFs can contain [unstructured information](https://en.wikipedia.org/wiki/Unstructured_data) that does not have a pre-defined data model or is not organized in a pre-defined manner. They are typically text-heavy and may contain a mix of figures, dates and numbers.

With the majority of available tools very often you have to process the entire PDF document, having no option to limit the data extraction to a specific section where the most valuable data lies in. However, some PDF table extraction tools do just that. Sad to say that even if you are lucky enough to have a table structure in your PDF it doesn't mean that you will be able to seamlessly extract data from it.

For example, let’s take a look at the following text-based PDF with some fake content. It has quite noticeable and distinguishing (although borderless) rows and columns:

<https://monosnap.com/file/0k6bzVTxo111zpSelM4YrvLXIXdWWY>



With only minor inspection you could have missed one important pattern: text at the intersection of some rows and columns is stacked and shifted so that it could hardly be recognized as the additional feature of the same data row.

Nonetheless, any data that does not fit nicely into a column or a row is [widely considered](https://www.marklogic.com/blog/6-things-you-need-to-know-about-unstructured-content/) unstructured, we can identify this particular real-world phenomenon as semi-structured data.

Which does not make it easier to parse data from a given table for any out-of-box extracting algorithm. While those tools may have reasonably efficient results, in this particular case we require extra development effort to fit your requirements. Moving forward with this tutorial you'll find a non-trivial solution to this challenge.

### Scope of this tutorial

In this tutorial you will learn how to:

* [use out-of-box solutions to extract tables from PDF](https://render.githubusercontent.com/view/ipynb?commit=51143b0e19d1ffee8137959be468c2cebcb1e9be&enc_url=68747470733a2f2f7261772e67697468756275736572636f6e74656e742e636f6d2f776f6c64656d6172672f756e737472756374757265645f646174615f706f73742f353131343362306531396431666665653831333739353962653436386332636562636231653962652f6d61696e2e6970796e62&nwo=woldemarg%2Funstructured_data_post&path=main.ipynb&repository_id=304269480&repository_type=Repository#Out-of-box-solutions-for-table-extraction)
* [get a raw text from PDF with the authentic document layout](https://render.githubusercontent.com/view/ipynb?commit=51143b0e19d1ffee8137959be468c2cebcb1e9be&enc_url=68747470733a2f2f7261772e67697468756275736572636f6e74656e742e636f6d2f776f6c64656d6172672f756e737472756374757265645f646174615f706f73742f353131343362306531396431666665653831333739353962653436386332636562636231653962652f6d61696e2e6970796e62&nwo=woldemarg%2Funstructured_data_post&path=main.ipynb&repository_id=304269480&repository_type=Repository#Processing-raw-text-from-PDF-with-authentic-layout)
* [perform text manipulations with numpy and pandas](https://render.githubusercontent.com/view/ipynb?commit=51143b0e19d1ffee8137959be468c2cebcb1e9be&enc_url=68747470733a2f2f7261772e67697468756275736572636f6e74656e742e636f6d2f776f6c64656d6172672f756e737472756374757265645f646174615f706f73742f353131343362306531396431666665653831333739353962653436386332636562636231653962652f6d61696e2e6970796e62&nwo=woldemarg%2Funstructured_data_post&path=main.ipynb&repository_id=304269480&repository_type=Repository#Manipuating-text-data-with-numpy-and-pandas)

More generally you will get a sense of how to deal with context-specific data structures in a range of data extracting tasks.

### Out-of-box-solutions for table extraction

To affirm the truth of the above statements we'll try to parse our semi-structured data with ready-made Python modules, specially assigned to extract tables from PDFs. Among the most popular out-of-box algorithms are [camelot-py](https://camelot-py.readthedocs.io/en/master/) and [tabula-py](https://tabula-py.readthedocs.io/en/latest/tabula.html). They both showed themselves to be effective in many complicated contexts. Let's see how they meet our challenge:

tabula\_df = (tabula

.read\_pdf(PDF,

stream=True,

pages="all"))

camelot\_df = (camelot

.read\_pdf(PDF,

flavor="stream",

suppress\_stdout=True,

pages="all")

[0].df)

Seems like our initial choice has turned into a miserable failure! While [tabula-py](https://tabula-py.readthedocs.io/en/latest/tabula.html) appears to be slightly better in detecting a grid layout of our table, it still provides a lot of extra work to split the text in a second column, not saying it has completely kicked off the last 'hanging' row of the original table:

<https://monosnap.com/file/NwvtcuCOq7M5v892S7LAAjXUY30Uip>

As to the output of [camelot-py](https://camelot-py.readthedocs.io/en/master/) - it is obvious that all the relevant information about interposition of text in columns has been completely vanished:

<https://monosnap.com/file/lThxMD80UbeWAu4cGPhl6jK6fitqgF>



From this moment on, we'll proceed with building our custom parsing algorithm.

### Processing raw text from PDF with an authentic layout

To begin with, we need a basis for our custom algorithm to work on. This should be a string input, fully representing the layout of the original document. Again you have quite a few options (think of python modules) to choose from. In our tutorial, we'll decide on [pdfminer](https://pdfminersix.readthedocs.io/en/latest/) and [pdftotext](https://github.com/jalan/pdftotext) to experiment with. Here is the output of their work:

pdfminer\_string = StringIO()

with open(PDF, "rb") as in\_file:

parser = PDFParser(in\_file)

doc = PDFDocument(parser)

rsrcmgr = PDFResourceManager()

device = TextConverter(rsrcmgr,

pdfminer\_string,

laparams=LAParams())

interpreter = PDFPageInterpreter(rsrcmgr, device)

for page in PDFPage.create\_pages(doc):

interpreter.process\_page(page)

pdfminer\_lines = pdfminer\_string.getvalue().splitlines()

pdfminer\_lines = [ln for ln in pdfminer\_lines if ln]

with open(PDF, 'rb') as file:

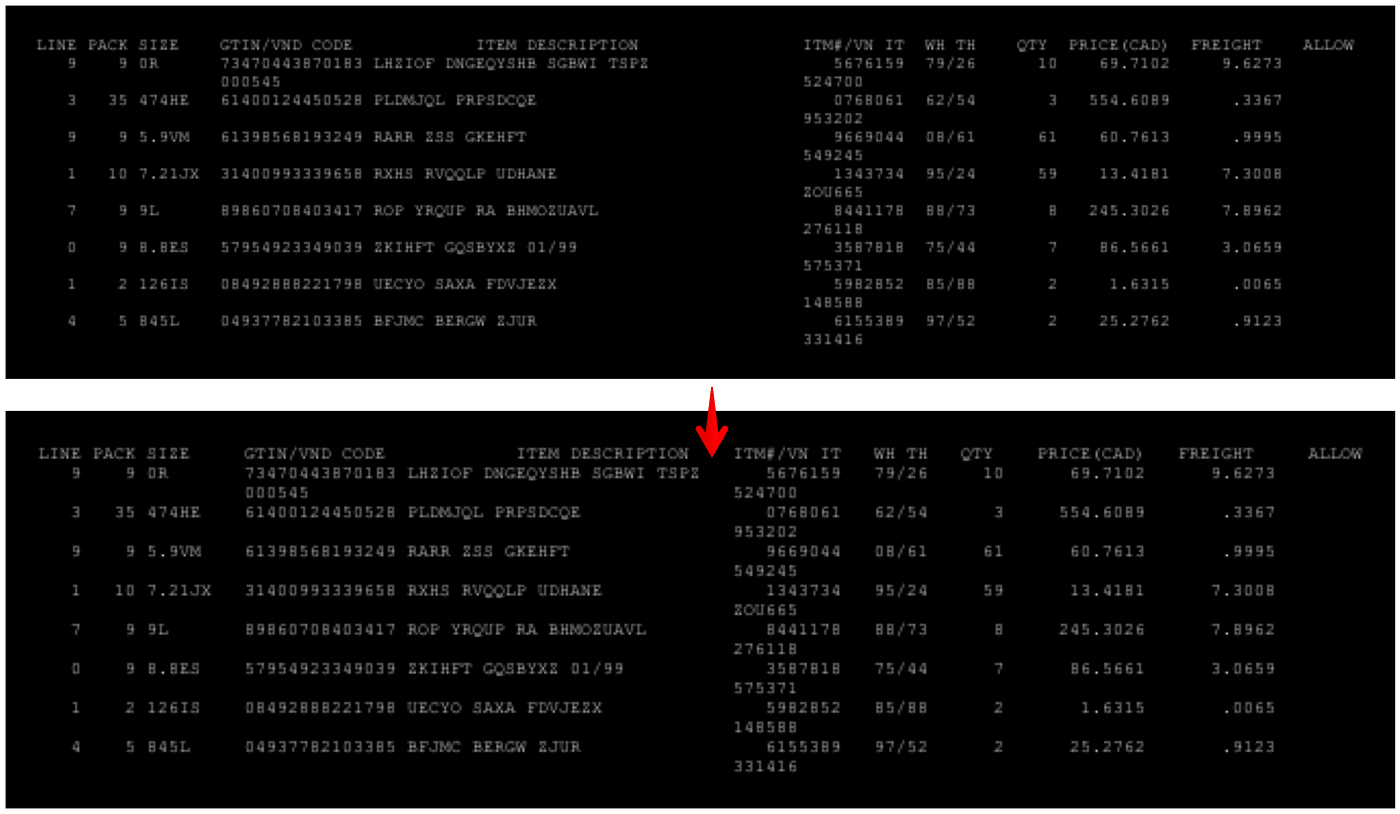
pdftotext\_string = pdftotext.PDF(file)

pdftotext\_lines = ("\n\n".join(pdftotext\_string).splitlines())

pdftotext\_lines = [ln for ln in pdftotext\_lines if ln]

It looks like both modules have produced quite satisfactory results, except for [pdftotext](https://github.com/jalan/pdftotext) has stripped some whitespace between 5th and 6th arbitrary columns:

<https://monosnap.com/file/NiYo5y799S4XZXQg0GWWbQvU4Gce33>



But the most essential is that in both cases we were able to preserve visible intervals between blocks of text aligned vertically (hello monospace font!)

### Manipulating text data with numpy and pandas

Another good news is that those blank spaces are somewhat that make you perceive vertically aligned text as a single whole element, i.e. a column in our table layout! This is a Law of Proximity in action. If you have never heard about Gestalt Theory of Visual Design before, please refer to the [link](https://www.creativebloq.com/graphic-design/gestalt-theory-10134960). What you should understand in particular is that the Law of Proximity is a long way off from any law of physics. It is just in your mind, so each time you have to explicitly tell a computer how close an arrangement of elements should be to have proximity. In our case, it is enough to have a straight vertical line of minimum one whitespace in width to evenly separate columns from each other. All we need is to show our custom algorithm where those whitespace-line dividers are.

#### Turning string input into a char matrix

To do the trick we'll turn our string output from [pdfminer](https://pdfminersix.readthedocs.io/en/latest/) into a char matrix, i.e. a structure with each string element including whitespaces in their separate cells. With some helper functions in the following block of code we sequentially perform the transformation and searching for above mentioned dividers:

pdf\_lines = pd.Series(pdfminer\_lines)

max\_width = pdf\_lines.str.len().max()

pdf\_lines\_adjusted = pdf\_lines.apply(adjust\_string, args=(max\_width, ))

pdf\_lines\_mtx = np.stack(pdf\_lines\_adjusted.map(list).to\_numpy())

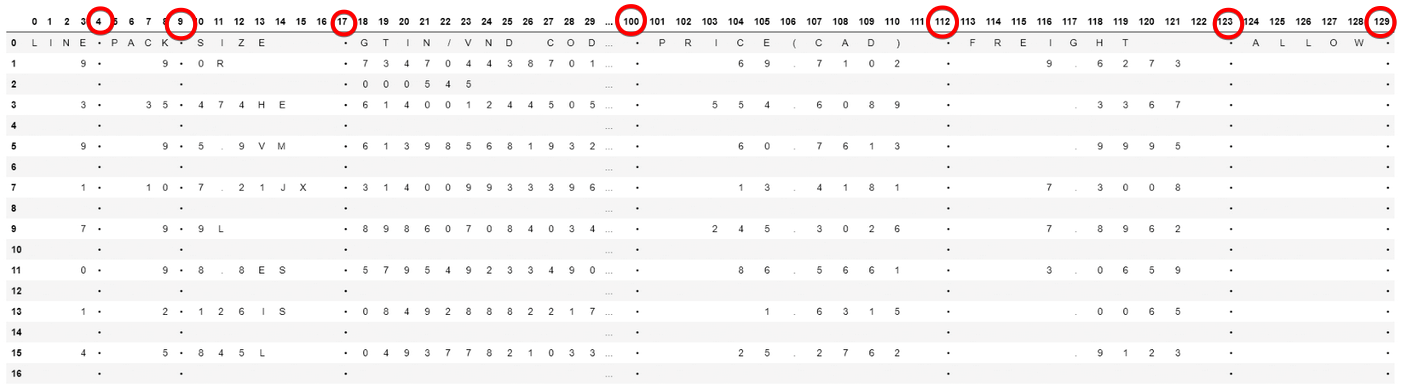
whitespace\_сols = np.where(np.all(pdf\_lines\_mtx == ' ', axis=0))[0]

rightmost\_whitespace\_cols = np.where(np.diff(whitespace\_сols) != 1)[0]

column\_dividers = np.append(whitespace\_сols[rightmost\_whitespace\_cols],

whitespace\_сols[-1])

<https://monosnap.com/file/LO65mxfK53vwAUxL2XGHdPy04kUzU6>



#### Splitting raw text into columns

Seems like we are ready to move forward and perform splitting of input lines into newly defined columns:

splitted\_lines = []

for row in pdf\_lines\_adjusted:

current\_row = []

for e in range(len(column\_dividers) - 1):

if e == 0:

current\_row.append(row[: column\_dividers[e]].strip())

(current\_row

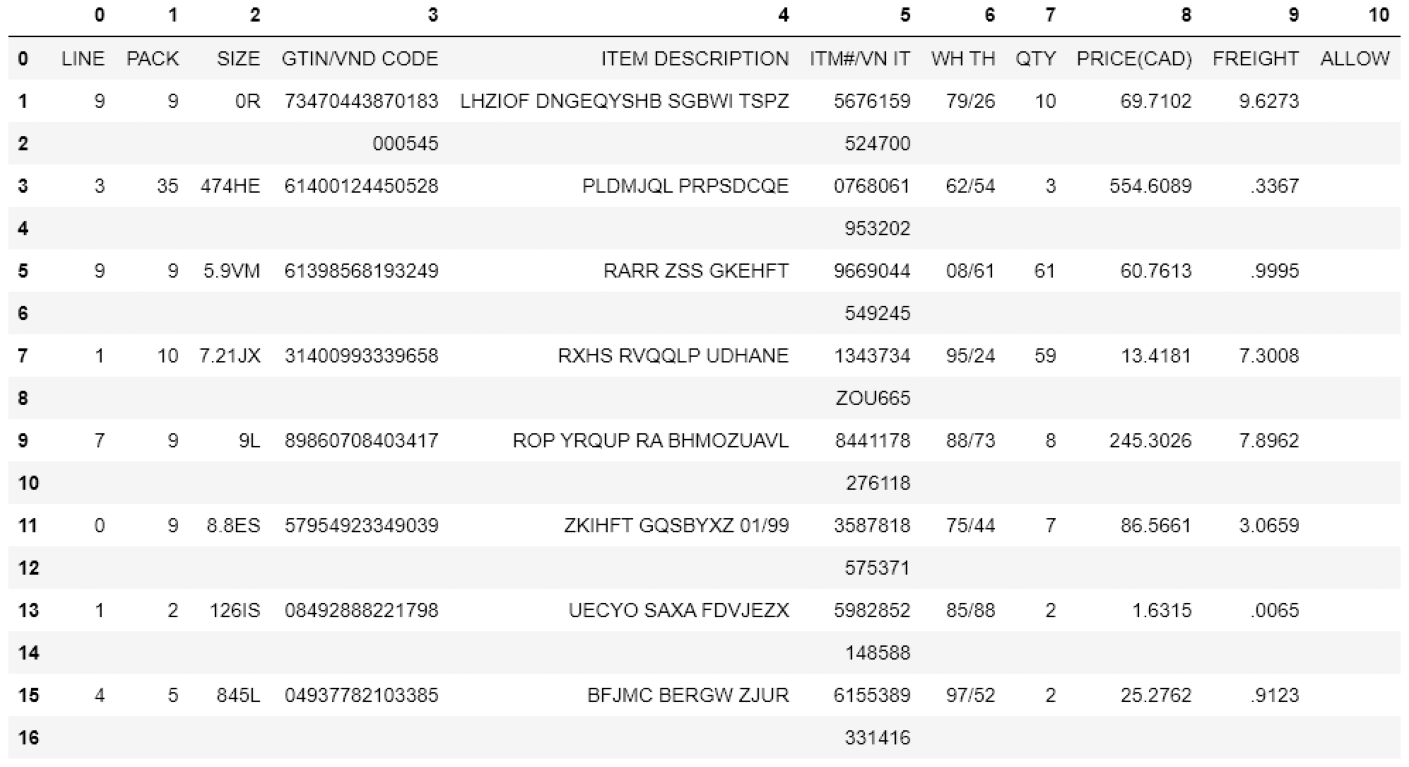
.append(row[column\_dividers[e]: column\_dividers[e + 1]]

.strip()))

current\_row.append(row[column\_dividers[-1]:].strip())

splitted\_lines.append(current\_row)

<https://monosnap.com/file/CI6S3MvWRNekyMY2FiXgDCHNvklpuW>



#### Dealing with stacked data in rows

Amazing! Our next step will be to move those stacked text-blocks within 'hanging' (even) rows to the right and up onto (⤴) the same level with a correspondent odd rows where they authentically belong to. Alongside we'll separate the first row as the header of our table to deal with it individually:

header\_row = pdf\_df.iloc[0, :].copy()

table\_rows = pdf\_df.iloc[1:, :].copy()

odd\_rows = table\_rows.iloc[::2].copy()

odd\_rows.reset\_index(drop=True, inplace=True)

evn\_rows = table\_rows[1::2].copy()

evn\_rows.reset\_index(drop=True, inplace=True)

evn\_rows.replace({'': np.nan}, inplace=True)

evn\_rows.dropna(axis=1, how='all', inplace=True)

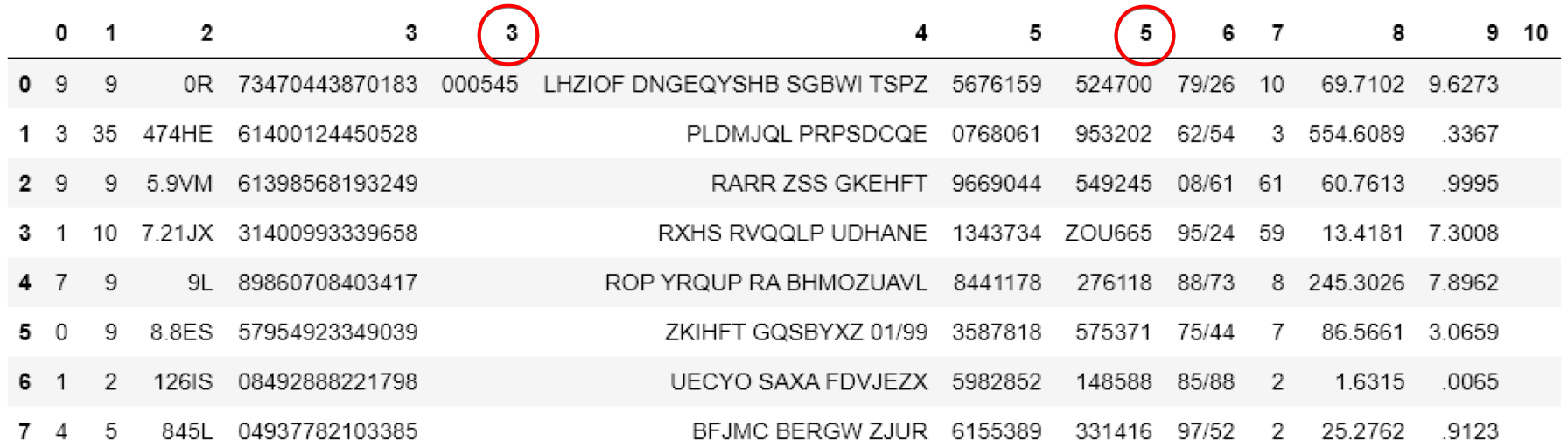
all\_rows = pd.concat([odd\_rows, evn\_rows], axis=1, sort=True)

all\_rows.sort\_index(axis=1, inplace=True)

#### Matching columns to original headers

Have you noticed how naturally and consistently our table got doubling headers for additional columns, such as 3-3 and 5-5?

<https://monosnap.com/file/S7dPup1e8ZQrTePrEjhFpJgSQIWMIy>



To further refer all the columns to original headers of the table we'll perform the following renaming trick: we'll assign all the doubling entries new unique subtitles (postfixes) respect to their parent (leftmost) elements. Such as a sequence [0, 1, 2, 3, 3, 4...] will become [0, 1, 2, 3, 3\_1, 4...].

def add\_postfix\_after(in\_list):

out\_list = []

x = 0

while x < len(in\_list) - 1:

if in\_list[x] != in\_list[x + 1]:

out\_list.append(str(in\_list[x]))

x += 1

else:

out\_list.append(str(in\_list[x]))

y = 1

while x + y < len(in\_list):

if in\_list[x] == in\_list[x + y]:

out\_list.append("{}\_{}".format(in\_list[x], y))

y += 1

else:

break

x += y

if x == len(in\_list) - 1:

out\_list.append(str(in\_list[-1]))

return out\_list

all\_rows.columns = add\_postfix\_after(all\_rows.columns.astype(int).tolist())

<https://monosnap.com/file/h4lxEKtgyjRegi6ITYz2aUzeAegxut>



The somewhat similar transformation we'll perform on original headers of the table (being separated from data on the previous step). It is assumed you have already noticed one particular pattern there: all the columns with stacked text have '/' in their titles, which denotes the name of a column we've got from data unstacking. To match columns with their titles we need some kind of a lookup table for headers to refer to and perform the final touch:

header\_splitted = header\_row.str.split('/')

header\_indicies = []

for i, element in enumerate(header\_splitted):

if len(element) == 1:

header\_indicies.append(str(i))

else:

for j, w in enumerate(element):

if j == 0:

header\_indicies.append(str(i))

else:

header\_indicies.append('{}\_{}'.format(i, j))

header\_elements = [item for sublist in header\_splitted

for item in sublist]

column\_names = pd.Series(data=header\_elements,

index=header\_indicies)

all\_rows.columns = column\_names[all\_rows.columns]

<https://monosnap.com/file/o8PlawGaxKHMTlxe5k3UdxUDpERpdB>

