IQ Bot Custom Logic
User Guide

Requests to update the document can be sent to technicalmarketing@automationanywhere.com

Table of Contents

Introduction

As businesses search for additional means of efficiency and optimization, the role of automation becomes an increasingly clear enabler and competitive advantage. To achieve efficient end-to-end automation, it's necessary to make the system more capable so expert workers can remain focused on business-critical tasks instead of handling countless exceptions.

Streamlining data extraction from documents is one area ideal for automation as current systems have built-in rules and assumptions about how documents appear. Since it is impossible to account for all document variations, having the ability to add customized extraction rules can help improve results.

The custom logic feature in IQ Bot enables users to make precise changes to extractions results using Python scripts. Creating automatic fine-tuning and flexible adjustments on extracted data streamlines data integration into target systems, further reducing the need for human action in the process.

Using custom logic, a user can modify field or table extraction results in numerous ways, such as:

- Removing a specific word, number, symbol, or phrase
- Locating and extracting specific values from a string of text
- Applying a regex filter
- Deleting rows that contains a specific value
- Adding values to a new column if another column contains a specific value
- Querying a database and return data related to the extraction value
- Calling an external machine learning system to analyze text

The custom logic feature leverages the simplicity and power of Python code to provide nearly endless possibilities for refining extracted document data.

This document explains how to use the custom logic feature, details specifics of its capabilities and performance, and provides a multitude of Python code examples demonstrating how to implement common use cases.

Getting Started

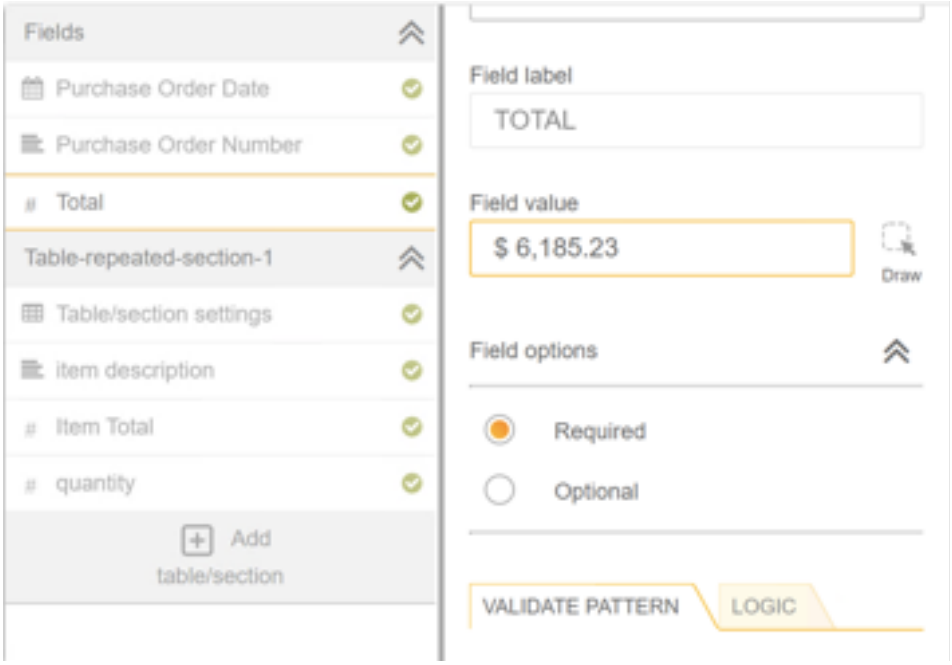
The custom logic feature may not be enabled by default. To enable the feature, add or edit a **features.json** file in the **Configurations** folder (by default C:\Program Files (x86)\Automation Anywhere IQ Bot\Configurations) and make sure that the attributes "`fieldLogic`", "`tableLogic`", and "`logicEditor:fullscreen`" are all set to `true`.

It should look something like this:

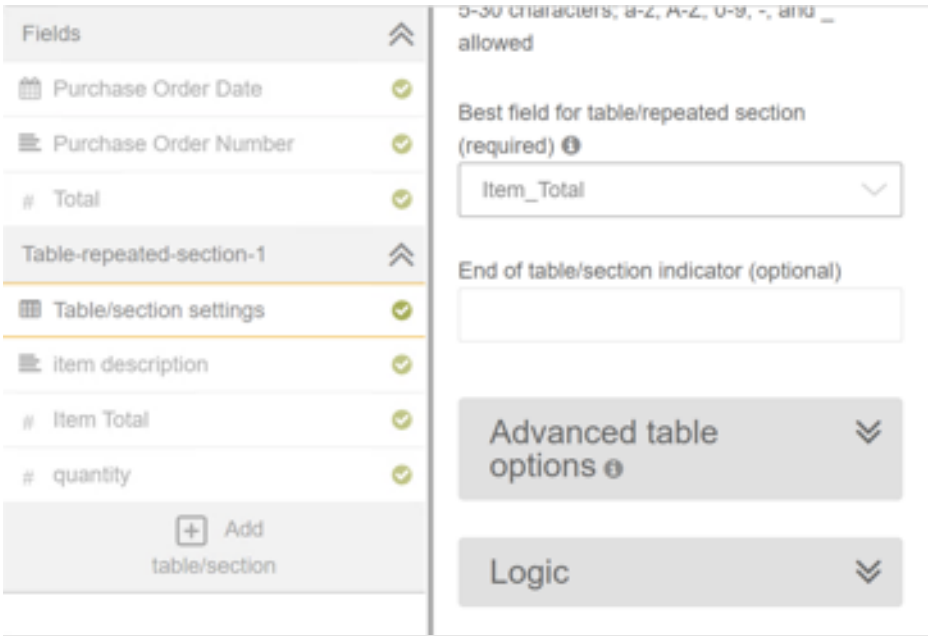
```
{
      …
      …
      "fieldLogic": true,
      "tableLogic": true,
      "logicEditor:fullscreen":true
}
```

To apply custom logic to an extracted field or table, you can click the **Logic** section found under either (A) the **Field options** section of an extracted field or (B) under the **Table/section settings** of an extracted table.

(A)



(B)



This will display a code box. In the code box, a user can input Python code to manipulate the extraction value, which is automatically saved to a Python variable with a default name. For individual fields, the extraction value is saved to a default variable called `field_value` and for tables, the table is saved to a default variable called `table_value`. The code box will output the final value of the default variable after all the code has been run.

Here a simple example of custom logic in action:

This single line of Python code will convert all lowercase letters to uppercase letters.

Example input:
`Julia McDaniel`
Resulting output:
`JULIA MCDANIEL`

This functionality is commonly used for standardization of extraction data prior to entry into a backend system. This helps to ensure stored data has consistent formats.

Other practical uses include converting a word format date to a numerical format date (e.g., "December 7, 2018" to "12/7/2018") or removal of symbols in extraction data (e.g., convert "$537.14" to "537.14"). These kinds of changes are extremely useful when sending extraction data to systems that require data to be in specific formats.

Feature Details

IQ Bot comes with Python version 3.5.4. The custom logic feature will execute the code using the Python version installed on the IQ Bot host system. The following additional Python packages are included:

Pre-installed packages:

arabic-reshaper (2.0.15): Reconstructs Arabic sentences to be used in applications that don't support Arabic script

certifi (2019.6.16): Root Certificates collection for validating the trustworthiness of SSL certificates while verifying the identity of TLS hosts

chardet (3.0.4): Detects languages from unknown character encodings

cx-Oracle (7.1.3): Accesses and interacts with Oracle databases

DateTime (4.3): Provides classes for manipulating dates and times

dateutils (0.6.6): Provides methods for manipulating dates and times

future (0.17.1): Allows cross-compatibility of code use between Python 2 and Python 3

idna (2.8): Supports the Internationalized Domain Names in Applications (IDNA) protocol

inflection (0.3.1): Allows for specific string manipulations such as singularizing and pluralizing words, and converting CamelCase to underscored string.

Jinja2 (2.10.1): Modern templating language for Python

MarkupSafe (1.1.1): Implements a text object that escapes characters for safe use of HTML and XML

numpy (1.16.4): Advanced scientific and mathematic operations

opencv-python (4.1.0.25): Contains open source computer vision algorithms

pandas (0.24.2): Provides flexible data structures for easier use and manipulation for structured data

Pillow (6.0.0): Provides extended image processing capabilities

pip (9.0.1): Allows for easy Python package installation and management

pymongo (3.8.0): Accesses and interacts with MongoDB databases

pyodbc (4.0.26): Accesses and interacts with ODBC databases

python-dateutil (2.8.0): Provides additional functionality for manipulating dates and times beyond the "DateTime" library

pytz (2019.1): Works with time zones

requests (2.22.0): Allows user to send HTTP requests

setuptools (28.8.0): Facilitates packaging of Python projects

six (1.12.0): Provides functions for creating Python code compatible with both to Python 2 and 3

urllib3 (1.25.3): Alternate library for allowing user to send HTTP requests

zope.interface (4.6.0): Assists with labeling objects as conforming to a given API or contract

New packages can be installed from the command line using the command:

```
pip install[package-name]
```

Any new packages manually installed or updated will be accessible by IQ Bot custom logic.

Each custom logic code block for each field or table runs in a sequential order during extraction. Depending on nature of the code, custom logic may affect the rate at which documents are processed by IQ Bot.

Each custom logic code block is allowed to run for a maximum of 4 minutes. Any custom logic that runs beyond this threshold will trigger a timeout event that prevents IQ Bot server from hanging indefinitely. A timeout event may cause the document to go unprocessed.

Field Extraction Example Use Cases

This section will cover how to perform various string operations in Python to obtain a desired result from an extracted field value. A string in Python is a sequence of characters than can be manipulated in a variety of ways, including splitting, truncating, and replacing.

Replacing/deleting substrings

Ex. 1) The `.replace(a,b)` function will replace every instance of `a` with `b` in a given string. This example will replace the string `"USD "` with `"$"`. This is useful for standardizing between different data formats.

Example:
```
field_value = field_value.replace("USD ", "$")
```

Sample input:
```
USD 537.14
```

Resulting output:
```
$537.14
```

Ex. 2) This example will replace the string `"USD "` with an empty string (""). This results in the removal of the string `"USD "`. This is commonly used for removing unwanted data from text strings.

Example:
```
field_value = field_value.replace("USD ", "")
```

Sample input:
```
USD 537.14
```

Resulting output:
```
537.14
```

Specifying substrings by index

A user can select a specific part of a string by selecting the index range of a substring. The index will specify the starting character position for where the new string should begin and the ending character position to stop. The first character of a string has index position 0, the second character has index 1 and so on. For example, the phrase "Hello World" has 11 positions. If we wanted to create a substring "World" the index range would be [6:10]

Position	0	1	2	3	4	5	6	7	8	9	10	
Phrase	H	e	l	l	o		W	o	r	l	d	

Ex. 1) A user can specify the substring of a field value with:
`field_value[beginningIndex:endingIndex]`

Example:
```
field_value = field_value[4:9]
```

Sample input:
```
USD 537.14 is the price
```

Resulting output:
```
537.14
```

Ex. 2) A user can also specify a portion of the string that extends to the end of the string, no matter the length of the string, by specifying the beginning index and leaving the ending index blank.

Example:
```
field_value = field_value[4:]
```

Sample input:
```
USD 537.14 is the price
```

Resulting output:
```
537.14 is the price
```

Splitting strings

The `.split()` function can be used to split the field value by any character, symbol, or number of choice into values into separate array indices. This splitting indicator is called a delimiter.

Ex. 1) In the example below, we will use the space (or `" "`) as the delimiter. The following line of code will split the string at every occurrence of the delimiter and place each segment split by the delimiter into its own array element. The string with then be replaced with the resulting array, allowing the user to specify parts of the string using an index number.

Example:
```
field_value = field_value.split(" ")
```

Sample Input:
```
USD 537.14 is the price
```

Index structure:

Index	0	1	2	3	4
Field_value [index]	USD	537.14	is	the	price

Resulting output:
```
['USD', '537.14', 'is', 'the', 'price']
```

In order to reference a specific item in the split array, one can place the index number of the desired element of the array in brackets after the split method.

Sample input:
```
field_value = field_value.split(" ")[1]
```

Resulting output:
```
537.14
```

Ex. 2) A user can also specify a number to limit how many times the string should be split. The code below divides the string at only at the first 3 spaces. Each division is considered its own index. The second 3 in the brackets specifies element with index number 3.

Example:
```
field_value = field_value.split(" ",3)[3]
```

Sample input:
```
The price is USD 537.14
```

Index Structure:

Index	0	1	2	3
Field_value [index]	The	price	is	USD 537.14

Resulting output:
```
USD 537.14
```

Replace value if string contains certain value

Here we are using a conditional statement to check if the field value contains a certain substring. If that string is found, we replace it.

In the following example we will look for the word "decreased" in the string. If it is there, a replace string operation will remove and replace the string "The price decreased by USD " with a negative sign.

Example:

```
if "decreased" in field_value:
     field_value = field_value.replace("The price decreased by
     USD ", "-")
```

Sample input:

```
The price dropped by USD 537.14
```

Resulting output:

```
-537.14
```

Example Use cases with Tables

This section will cover common ways a user can make changes to tables, including adding/removing columns, applying a string operation on a specific cell in the table, and applying regex operations to the table.

Manipulating table values is much easier using the pre-installed `pandas` Python library. It is recommended to perform table operations using this library due to its flexibility.

In your code, you can use the code below to import the `pandas` library:

```
import pandas as pd
```

The first requirement with working with pandas is to convert the `table_values` variable into a pandas data frame. Table variables in IQ Bot by default are stored in a dictionary format that looks something like this:

```
table_values =

[{'item_description': 'Apple',  'unit_price': '$0.60', 'quantity': '40'},
 {'item_description': 'Orange', 'unit_price': '$1.20', 'quantity': '20'},
 {'item_description': 'Banana', 'unit_price': '$0.80', 'quantity': '30'},
 {'item_description': 'Peach' , 'unit_price': '$1.00', 'quantity': '30'}]
```

Using the code below will convert the `table_values` variable into a `pandas` data frame object which will be called "`df`".

```
df = pd.DataFrame.from_dict(table_values)
```

After the table is converted to a data frame format, it will look like this:

```
df =

   item_description  unit_price   quantity
0  Apple               $0.60         40
1  Orange              $1.20         20
2  Banana              $0.80         30
3  Peach               $1.00         50
```

Once the `table_values` variable is converted into a pandas data frame object, we can begin to apply changes to the values in the table with greater ease.

After the values of `df` have been modified in the desired manner, it must be saved back to the `table_values` variable for IQ Bot to recognize the changes. This can be done with the following code:

```
table_values = df.to_dict()
```

The above example table has a header and four lines.

The diagram below shows specifically how the elements of table are organized after being converted into a pandas data frame object.

Data Frame Structure

		Column #		
		0	1	2
		item_descripton	unit_price	quantity
Row Index	0	Apple	$0.60	40
	1	Orange	$1.20	20
	2	Banana	$0.80	30
	3	Peach	$1.00	50

Note that an additional column is added before the first. This column contains the indexes that corresponds to each row of the table but does not have a column index itself. The `table_value` dictionary keys will automatically be used to define as the header of a data

frame table, and the values the keys correspond to will be populated in the key below. The header does not have a row index. The row index starts with the first line after the header and starts with a value of zero. Columns can be referenced with either the column index number or the column description.

In the following examples, we will show the table inputs and outputs in data frame format.

Extract specific rows

In these examples, we will extract out specific rows of data from an extracted table using pandas data frame operations.

This Python code will change data frame object, or `df`, to a table which contains only rows in the specified list of rows, with each row being separated by a comma using the `.iloc` operator.

Example:

```
import pandas as pd
df = pd.DataFrame.from_dict(table_values)
df = df.iloc[1,3]
table_values = df.to_dict()
```

Sample input:

```
   item_description  unit_price   quantity
0  Apple               $0.60         40
1  Orange              $1.20         20
2  Banana              $0.80         30
3  Peach               $1.00         50
```

Resulting output:

```
   item_description  unit_price   quantity
1  Orange              $1.20         20
3  Peach               $1.00         50
```

Extract a specific range of rows

Ex. 1) Instead of using a comma to specify individual rows, a user can use a colon operator to denote a range of indices.

Note that in a pandas data frame, the first number specifies the first index of the range, but the second number specifies the number **after** the last index of the range. Thus, a range of `[0:3]` refers to rows 0, 1, and 2.

Example:

```
import pandas as pd
df = pd.DataFrame.from_dict(table_values)
df = df.iloc[1:3]
table_values = df.to_dict()
```

Sample input:

```
   item_description  unit_price   quantity
0  Apple               $0.60         40
1  Orange              $1.20         20
2  Banana              $0.80         30
3  Peach               $1.00         50
```

Resulting output:

```
   item_description  unit_price   quantity
1  Orange              $1.20         20
2  Banana              $0.80         30
```

Ex. 2) A user can also leave either the first or last index blank in order to select all rows of the data frame before or after the specified index.

Example:

```
import pandas as pd
df = pd.DataFrame.from_dict(table_values)
df = df.iloc[:2]
table_values = df.to_dict()
```

Sample input:

```
   item_description  unit_price   quantity
0  Apple               $0.60         40
1  Orange              $1.20         20
2  Banana              $0.80         30
3  Peach               $1.00         50
```

Resulting output:

```
   item_description  unit_price   quantity
0  Apple               $0.60         40
1  Orange              $1.20         20
```

Extract specific columns

Ex. 1) To extract the columns, a user can specify the column indices inside another pair of brackets, next to a blank row indicator.

Example:

```
import pandas as pd
df = pd.DataFrame.from_dict(table_values)
df = df.iloc[:,[0,2]]
table_values = df.to_dict()
```

Sample input:

```
   item_description  unit_price   quantity
0  Apple               $0.60         40
1  Orange              $1.20         20
2  Banana              $0.80         30
3  Peach               $1.00         50
```

Resulting output:

```
   item_description  quantity
0  Apple               40
1  Orange              20
2  Banana              30
3  Peach               50
```

Ex. 2) A user can alternately refer to the columns by their dictionary key names. The `.iloc` operator does not need to be used.

```
import pandas as pd
df = pd.DataFrame.from_dict(table_values)
df = df[["item_description","quantity"]]
table_values = df.to_dict()
```

Sample input:

```
   item_description  unit_price   quantity
0  Apple               $0.60         40
1  Orange              $1.20         20
2  Banana              $0.80         30
3  Peach               $1.00         50
```

Resulting output:

```
  item_description  quantity
0  Apple               40
1  Orange              20
2  Banana              30
3  Peach               50
```

Extract specific rows and columns

We can combine the previous examples to select specific rows and columns simultaneously for extraction. We can use the comma to specify a list of indices or the colon to specify a range of indices.

This code below will assign the rows indexed `1` and `3` under columns indexed `0` and `1` as the new data frame value of `df`:

```
import pandas as pd
df = pd.DataFrame.from_dict(table_values)

df = df.iloc[[1,3],[0:2]]
table_values = df.to_dict()
```

Example input:

```
   item_description  unit_price   quantity
0  Apple               $0.60         40
1  Orange              $1.20         20
2  Banana              $0.80         30
3  Peach               $1.00         50
```

Resulting output:

```
   item_description  unit_price
1  Orange              $1.20
3  Peach               $1.00
```

Extract a specific cell

In certain situations, a user may need to extract out a particular value from a complex table.

Ex. 1) A user can simply specify the cell using the row index and column index with the `.iloc` operator.

```
import pandas as pd
df = pd.DataFrame.from_dict(table_values)
df = df.iloc[2,0]
table_values = df.to_dict()
```

Example input:

```
   item_description  unit_price   quantity
0  Apple               $0.60         40
```

```
1   Orange              $1.20          20
2   Banana              $0.80          30
3   Peach               $1.00          50
```

Resulting output:

```
Banana
```

Ex. 2) Alternatively, a user can specify the column by its key name instead of its index with the `.loc` operator:

```
import pandas as pd
df = pd.DataFrame.from_dict(table_values)
df = df.loc[2, "item_description"]
table_values = df.to_dict()
```

Example input:

```
   item_description  unit_price   quantity
0  Apple               $0.60          40
1  Orange              $1.20          20
2  Banana              $0.80          30
3  Peach               $1.00          50
```

Resulting output:

```
Banana
```

Add a row to a table

This code adds a row in which all the current column names are specified as keys and new row entries are specified as key values in a Python dictionary object. This object is added to the table with the `.append()` method.

```
import pandas as pd
df = pd.DataFrame.from_dict(table_values)
df = df.append({'item_description': 'Watermelon',
                'unit_price': '$5.00',
                'quantity': '4'}, ignore_index=True)
table_values = df.to_dict()
```

Example input:

```
   item_description  unit_price   quantity
0  Apple               $0.60          40
1  Orange              $1.20          20
2  Banana              $0.80          30
```

```
3   Peach                $1.00          50
```

Resulting output:

```
   item_description  unit_price   quantity
0  Apple                $0.60          40
1  Orange               $1.20          20
2  Banana               $0.80          30
3  Peach                $1.00          50
4  Watermelon           $5.00           4
```

Delete rows that are missing values of a certain column

This Python code will remove any row in which column `item_total` is empty.

```
import pandas as pd
df = pd.DataFrame.from_dict(table_values)
df = df[(df["unit_price"] != "")]
table_values = df.to_dict()
```

Example input:

```
   item_description  unit_price   quantity
0  Apple                $0.60          40
1  Orange               $1.20          20
2  Banana                              30
3  Peach                $1.00          50
```

Resulting output:

```
   item_description  unit_price   quantity
0  Apple                $0.60          40
1  Orange               $1.20          20
2  Peach                $1.00          50
```

Delete rows matching a regular expression

This code will match a regular expression with each value under column `item_description`. If regex match (in this case, a string with "Item" followed by four numbers) delete the row containing that value. This code also imports the regex `re` library to use a regex function.

Example:

```
import pandas as pd
import re
```

```
df = pd.DataFrame.from_dict(table_values)

def is_found(string):
    a = re.findall('Item [0-9]{4}',string)
    if a:
        return False
    else:
        return True

df = df[(df["item_description"].apply(is_found))]
table_values = df.to_dict()
```

Sample input:

```
   item_description  unit_price   quantity
0  Apple               $0.60         40
1  Orange              $1.20         20
2  Item 0034           $0.80         30
3  Peach               $1.00         50
```

Resulting output:

```
   item_description  unit_price   quantity
0  Apple               $0.60         40
1  Orange              $1.20         20
2  Peach               $1.00         50
```

Replace all instances of a value of a specific column with another value

This code replaces all dollar signs with Euro symbols.

Example:

```
import pandas as pd
import re

df = pd.DataFrame.from_dict(table_values)
df["Item_Total"] = df["Item_Total"].replace({'$':'€'},
regex=True)
table_values = df.to_dict()
```

Sample input:

```
   item_description  unit_price   quantity
0  Apple               $0.60         40
1  Orange              $1.20         20
2  Banana              $0.80         30
3  Peach               $1.00         50
```

Resulting output:

```
   item_description  unit_price   quantity
0  Apple               €0.60         40
1  Orange              €1.20         20
2  Banana              €0.80         30
3  Peach               €1.00         50
```

Replace all instances of a value anywhere in the table with another value

This code will replace every instance of "`0range`" with "`Orange`" (zero with the letter "O") within the data frame, regardless of row or column. One can use the `.applymap()` method to apply any function to the all the individual cell values of the data frame. This is useful in handling exceptions.

Example:

```
import pandas as pd
import re

df = pd.DataFrame.from_dict(table_values)
def find_and_replace(value):
     return value.replace("0range","Orange")

df=df.applymap(find_and_replace)
table_values = df.to_dict()
```

Sample input:

```
   item_description  unit_price   quantity
0  Apple               $0.60         40
1  0range              $1.20         20
2  Banana              $0.80         30
3  Peach               $1.00         50
4  0range Juice        $3.00         8
```

Resulting output:

```
   item_description  unit_price   quantity
0  Apple               $0.60         40
1  Orange              $1.20         20
2  Banana              $0.80         30
3  Peach               $1.00         50
4  Orange Juice        $3.00         8
```

Add values to a column if other column matches a regular expression

This code will match a regular expression with each value under column `unit_price`. If the value contains a dollar sign, it will populate the corresponding `currency` value as "`USD`" .

Example:

```
import pandas as pd
import re

df = pd.DataFrame.from_dict(table_values)
def is_USD(string):
    a= re.findall('$*',string)
    if a:
        return True
    else:
        return False

df.loc[(df["unit_price"].apply(is_found)),"currency"] = 'USD'
table_values = df.to_dict()
```

Sample input:

```
   item_description  unit_price   quantity   currency
0  Apple               $0.60         40
1  Orange              $1.20         20
2  Banana              €0.80         30
3  Peach               $1.00         50
4  Watermelon          $5.00          4
```

Resulting output:

```
   item_description  unit_price   quantity   currency
0  Apple               $0.60         40         USD
1  Orange              $1.20         20         USD
2  Banana              €0.80         30
3  Peach               $1.00         50         USD
4  Watermelon          $5.00          4         USD
```

Using Python Libraries with custom logic

There are a multitude of other things a user can do using Python's extensive library collection. A user can import a library with the code `import [library-name]`. Here are a few ways to use them in manipulating fields:

Applying regex filters using the regex library

A user can search for a string of a specific kind of format using a regular expression, or regex. Regex is a string of characters that represents the format of a string a user wants to search for. The regex string can be adjusted to search for strings with certain characters or numbers in certain positions, strings of an exact length, or for strings that have certain range of numbers following specified characters. Import the regex library or `re` to apply regex with Python code.

These lines of code takes in a regex string, which is this case is `'([0-9]{3}-[0-9]{3}-[0-9]{4})'`, to search for any numerical strings in an XXX-XXX-XXXX format, such as phone numbers or social security numbers. If no matches are found, the original string is returned.

Example:

```
import re
def find_numbers(string):
    match = re.findall('([0-9]{3}-[0-9]{3}-[0-9]{4})',string)
    if match:
        return match
    else:
        return string

field_value = find_numbers(field_value)
```

Sample input:

```
His phone number is 222-444-8888, SSN is 123-456-7890, and DL#
is 1234567890
```

Resulting output:

```
['222-444-8888', '123-456-7890']
```

Reformat dates with the datetime library

The `datetime` library makes it easy to change the formats of dates.

Example:

```
from datetime import datetime
field_value = datetime.strptime(field_value, '%d %
%Y').strftime("%Y/%m/%d")
```

Sample input:
```
22 Aug 2016
```

Resulting output:
```
2016/08/22
```

Using external calls / APIs with custom logic

There are various external systems custom logic can utilize to apply modifications to data, such as REST services, databases, and language processing engines. The following outlines a few examples of how this could be utilized to improve a user's extraction results and capabilities.

Make an HTTP Request

This example demonstrates and HTTP request being made to an external address parser application. The address parser will return a parsed address for use. We will capture only the 'road' value of the response.

Example:
```
field_value = "818 Lexington Ave, #6, PO Box 1234, Brooklyn
11221 NY, USA"

import requests

url = "http://example-url.com/parser"

payload = "{\"query\": \""+field_value+"\"}"
headers = {
    'Content-Type': "application/json",
    'Accept': "*/*",
}

response = requests.request("POST", url, data=payload,
headers=headers)

resp = eval(response.text)
Adr = {}
for idic in resp:
    Adr[idic['label']] = idic['value']

field_value = Adr['road']
```

Sample input:

```
818 Lexington Ave, #6, PO Box 1234, Brooklyn 11221 NY, USA
```

Resulting output:

```
Lexington Ave
```

Call an external machine learning system to intelligently identify text

Calling external machine learning systems is another way a user can utilize custom logic. With an external machine learning system, a user can apply a variety of complex actions on unstructured text such as recognize client names, classify the intention of a human message, and automatically identify and translate foreign languages.

The code below is one such example that uses an API call to send text to a machine learning system that identifies serial numbers in the text, regardless of the position, format, or length of the serial number. The system extracts data at a level of intelligence beyond standard regex or string manipulation.

Using external services, such as machine learning, unlocks a new dimension of capability in custom logic to perform advanced operations on extracted document data.

More information about using machine learning in custom logic can be found in this video: https://automationanywhere.wistia.com/medias/czg16jtvvw

Note: Code for setting up and running a machine learning system is not included.

```
import pandas as pd
import requests

df = pd.DataFrame.from_dict(table_values)
RawBody = df.loc[:, "Raw_Body"][0]

url = "http://localhost:5000/models"

model = "ML-model-01"

payload = "{\"text\": \""+RawBody+"-",\"model\":\""+model+"\"}"
headers = {
     'Content-Type': "application/json",
     'Accept': "*/*",
}
```

```
response = requests.request("POST", url, data=payload,
headers=headers)

resp = eval(response.text)
for ent in resp['entities']:
     ENT_TYPE = ent['label']
     ENT_SCORE = ent['score']
     ENT_TEXT = end['text']

df.loc[: ,"Order_List"][0] = ENT_TEXT


table_values = df.to_dict()
```

Sample input:

```
I am looking for the corresponding product name for product
number ZLS-539AJ297. Can you help me?
```

Resulting output:

```
ZLS-539AJ297
```

Query database to get corresponding info

This code will take a vendor name and return its corresponding vendor ID. Note: the following example's database info is not representative of a real database environment.

```
import pyodbc
conn = pyodbc.connect('Driver={SQL Server};'
                      'Server=localhost\sqlexpress;'
                      'Database=TestDB;'
                      'Trusted_Connection=no;'
                      'uid=user123;'
                      'pwd=pass123')

cursor = conn.cursor()
cursor.execute('SELECT vendor_id FROM
[TestDB].[dbo].[VENDOR_INFO_1] where vendor_name_short =
\''+field_value+'\'')

for row in cursor:
    field_value = row[0]
```

Sample input:

```
Adego Industries
```

Resulting output:

XIS77823964

Query database to get fuzzy match

This example will take a vendor name and check its closest match to a list of vendors on a separate text file. This will reduce any OCR inaccuracies.

```
from fuzzywuzzy import fuzz,process

text_file = open("c:\\vendorlist.txt", "r")
options = text_file.read().split(',')
text_file.close()

print(options)

Ratios = process.extract(field_value,options)
highest = process.extractOne(field_value,options)

field_value = highest[0]
```

Example input:
Adeg0 Industries
Resulting output:
Adego Industries