Improving Accuracy & Efficiency in Document Handling for Business Processes

Alison Major
Computer Science
Lewis University
Romeoville, Illinois, USA
AlisonMMajor@lewisu.edu

***Abstract*—Data is found in a variety of digital formats: email messages, spreadsheets, images, sound files. There are a number of tools that have been designed to assist in automatically extracting data from certain formats. Optical Character Recognition (known as OCR) can be used to convert images into text, though not always in a structured and useful format. Natural-language processing allows computers to extract information from text written in the natural language form. Is there a method to allow multiple forms of information to be received by a system and automatically pull the desired data from those different formats? For example, if a customer submits a purchase order as a scanned image that we want to automatically pull item numbers from, what is the ideal method for processing that file and information? Similarly, if a customer submits an order in an email message in prose or with an embedded table, what are our options for extracting the desired information? This paper will review technologies that use OCR, natural-language processing, and machine learning to find optimal options for extracting data from a variety of inputs.**

***Index Terms*—optical character recognition (OCR), artificial intelligence (AI), machine learning ML), natural language processing (NLP), robotic process automation (RPA)**

I. Introduction

Many companies receive input from users in a variety of ways. Some may email, some send attached files. There may be a web application used to submit forms of information or capture a photo of a document on their phone. Technology has provided us with ample options to submit information.

For a human to process submitted information, they need to read the data in whichever format it is received, then understand it enough to enter it into the appropriate system. This basic decision-making is generally not a difficult task, but when needed at a large scale, can become not only tiresome but prone to entry errors.

In a controlled study in 2009 at UNLV, a set of 215 students were provided with 30 datasheets that contained six types of data to process. When the students only used visual confirmation of the correctness of the data, they made an average of 10.23 errors. Progressive steps to automate confirming information improved the error count [1]. Data can be cheaply entered in by hiring large numbers of low-cost employees, but more can be done to bring not only the data to a cleaner level, but also provide efficient methods for the people involved.

To demonstrate challenges and considerations in how to process a variety of inputs, we'll review the challenges and some data behind the manual entry of information (Section II). With the comprehension of the problem at hand, we will define optical character recognition (OCR) and the challenges involved with this method (Section III). Built on the foundation of the business flow for manual data entry and the use of OCR, we will present a flow diagram for processing data (Section IV). This section will break down these steps for processing, define document classification types, and review the steps needed to handle the data in a repetitive way.

We'll focus on receiving orders and adding them to a company's ordering system. Ideal situations would see that customer orders be received in a consistent format. However, when there are a large number of customers, or if some customers carry a lot of weight, they will stick with the process that is easiest for them. This means that whatever comes out of the systems and applications that they use is what you will need to work from.

There are many solutions already available, depending on the format of the document received and the type of data being handled. We will then explore combining this flow for a number of document types into a single pipeline, review how PepsiCo successfully used this method, and explore how others may do likewise (Section V).

II. Manual Data Entry

In 2018, Goldman Sachs reported that the direct and indirect cost of manual data entry for global businesses was estimated to be about $2.7 trillion [2]. Gartner's study in 2019 found that avoidable rework in accounting departments amounted to 30% of a full-time employees' time; for an accounting staff of 40 full-time employees, this amounts to 25,000 hours per year and about $878,000 [3]!

While modern applications provide us with many conveniences, many companies or industries have legacy processes in place. People deal with paper forms and documents every day, which then must be typed into digital systems. Employees will receive emails, faxes, and files that they must then determine which type of document it is and then process it accordingly for that customer, usually by typing the data contained into another system. This manual data entry leaves room for error, as Gartner found. Additionally, manual entry is very time-consuming; talented employees could better spend their time on tasks that provide real value to building the customer relationship and improving the customer experience, rather than focusing on wearying data entry.

III. What is OCR?

Optical Character Recognition, more commonly known as OCR, is a technique that results in converting images and files into machine-readable data. The technology reviews images for fonts and shapes, matching information with text that can be reviewed and stored.

While providing actionable information from different types of digital files, there are still many challenges with OCR. Images and pages may be received with incorrect orientation or skewed at an angle (imagine the paper is fed crookedly into a fax or scanner, or a picture taken with a phone that is not perfectly aligned). There may be noise or distortion on the page, or handwriting and other marks obscuring the printed text. If background colors or images are present, these can also impact the technology's ability to get a clear read on the information.

Some of these challenges can be overcome through the use of preprocessing. Pages can be scaled and adjusted to be the correct orientation. Filters can be applied to clean up noise. Newer iterations of OCR technologies also apply machine learning (ML) that can be used to better identify text characters when the image is unclear.

Introducing OCR to the process of receiving files from customers gives us a digital version of the data. Rather than an employee typing the information character-for-character, output from an OCR engine provides the employee with data that can be copy-pasted with minor corrections. While small, this technique introduces some new efficiencies that we can leverage in an improved data processing flow.

IV. Processing Data

Let's say we run a ball factory that produces balls of different colors and sizes. We receive orders from our customers in a number of ways. One of our larger customers has its own inventory system, so when they order balls, they enter into their system and a templated spreadsheet is generated. This customer then emails us this file as an attachment. Another customer runs a small shop where the manager prints out an order request from their own template in a word processing application that the owner captures a picture of from their phone and sends along.

In the ideal setting, we'd force all of our customers to use a standardized process that is easy for us to ingest through simple rules. We could even create some macros in our spreadsheets to process the information and make it easy to enter into our own ordering system. However, most companies do not have this ability for a number of reasons. Instead, they must consider how to handle the different types of documents by creating a business flow like the one in "Fig. 1".

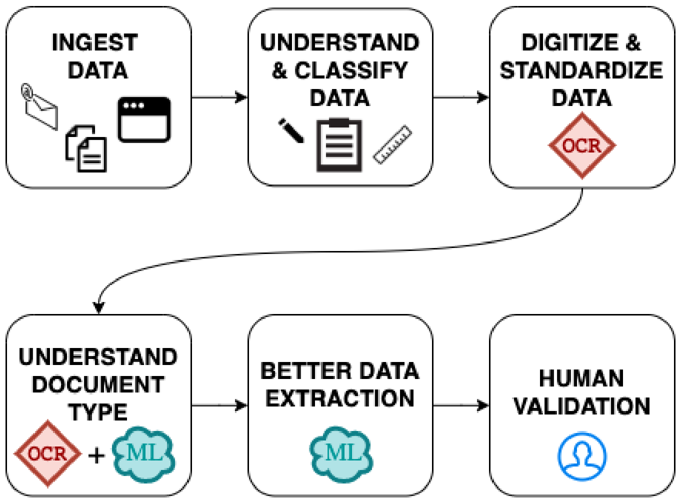


Fig. 1. Document flow in a generic business process for extracting data from different inputs.

A. Ingest Data

The first step in our flow involves ingesting data. This just means that we are receiving it somehow. Oftentimes, this ingestion step may just be an employee receiving an email with an order attached or in the body of the message.

Some companies have different ingestion methods that allow for integrated flow to their ordering system, like having users place orders by logging into a website. This is even more ideal, as integrated systems allow for little-to-no human error once the order has been entered by the customer. New businesses may have this option, but those that have been around longer may have customers that are used to a certain method or have their own legacy systems or large volumes of data that make manual entry into a web application difficult.

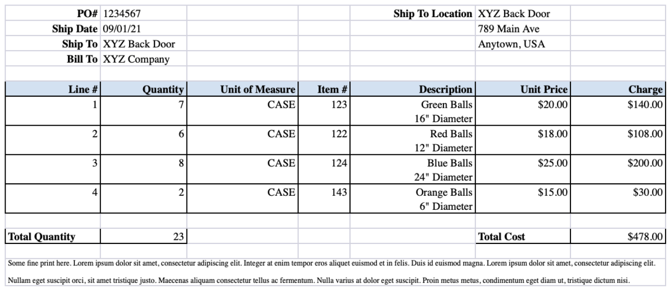
To continue in our journey, we'll assume that we are ingesting documents that may come in a variety of types and formats and are not entered into an integrated system. We will assume that our orders for our ball factory are currently manually entered by a team of people.

B. Understand & Classify Data

Since we are receiving orders in several file types (spreadsheet, pdf, email message, etc) that could be handled in different ways, we must find a method for understanding and classifying the data into one of three primary categories: Structured Data, Semi-Structured Data, and Unstructured Data. Once we have organized our documents, we can take the appropriate next action.

1) Structured Data: Often in the format of comma-separated values (CSV) and spreadsheets, structured data is formatted in a consistent template. The page contains tables and predictable positions on the page or in the document. These documents could also be pdfs or other file types; the common thread is that the information is in the same layout for every document received.

Because of its predictability, this type of data can be gathered through simple data extraction scripts or template-based OCR. Rule-based approaches work well because we always know where to expect information that can be linked to a key-value pair (like ”date: 8/14/2021” or ”PO: 1234567”).



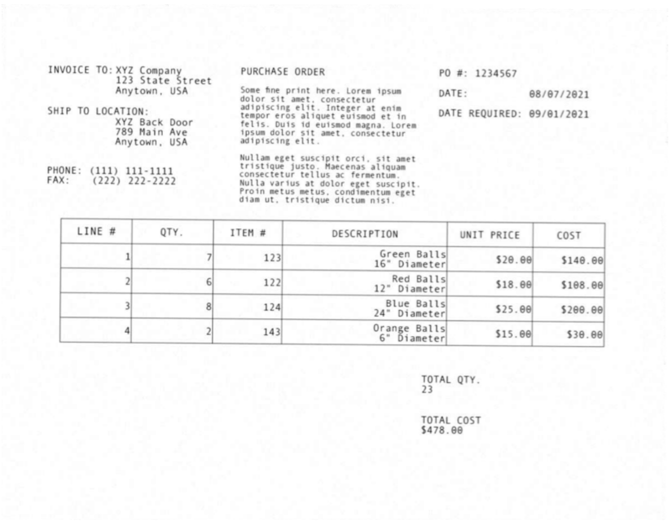
PO#	1234567			**Ship To Location**	XYZ Back Door	
Ship Date	09/01/21				789 Main Ave	
Ship To	XYZ Back Door				Anytown, USA	
Bill To	XYZ Company					
Line #	**Quantity**	**Unit of Measure**	**Item #**	**Description**	**Unit Price**	**Charge**
1	7	CASE	123	Green Balls 16" Diameter	$20.00	$140.00
2	6	CASE	122	Red Balls 12" Diameter	$18.00	$108.00
3	8	CASE	124	Blue Balls 24" Diameter	$25.00	$200.00
4	2	CASE	143	Orange Balls 6" Diameter	$15.00	$30.00
Total Quantity	23				**Total Cost**	$478.00

Some fine print here. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Integer at enim tempor eros aliquet euismod et in felis. Duis id euismod magna. Lorem ipsum dolor sit amet, consectetur adipiscing elit.

Nullam eget suscipit orci, sit amet tristique justo. Maecenas aliquam consectetur tellus ac fermentum. Nulla varius at dolor eget suscipit. Proin metus metus, condimentum eget diam ut, tristique dictum nisi.

Fig. 2. A templated spreadsheet for a purchase order.

Using our ball factory example, the spreadsheet in “Fig. 2” could be an example of a structured document, if all spreadsheets received follow the same layout. This structured data allows for scripts to be written that can map information by using the expected position of a value or with regular expressions to collect the value from the file. We’ve proven a sample of this in the simple Python code in Appendix A, with unit tests for the methods created (see Appendix B).



INVOICE TO: XYZ Company
123 State Street
Anytown, USA

SHIP TO LOCATION:
XYZ Back Door
789 Main Ave
Anytown, USA

PHONE: (111) 111-1111
FAX: (222) 222-2222

PURCHASE ORDER

Some fine print here. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Integer at enim tempor eros aliquet euismod et in felis. Duis id euismod magna. Lorem ipsum dolor sit amet, consectetur adipiscing elit.

Nullam eget suscipit orci, sit amet tristique justo. Maecenas aliquam consectetur tellus ac fermentum. Nulla varius at dolor eget suscipit. Proin metus metus, condimentum eget diam ut, tristique dictum nisi.

PO #: 1234567

DATE: 08/07/2021

DATE REQUIRED: 09/01/2021

LINE #	QTY.	ITEM #	DESCRIPTION	UNIT PRICE	COST
1	7	123	Green Balls 16" Diameter	$20.00	$140.00
2	6	122	Red Balls 12" Diameter	$18.00	$108.00
3	8	124	Blue Balls 24" Diameter	$25.00	$200.00
4	2	143	Orange Balls 6" Diameter	$15.00	$30.00

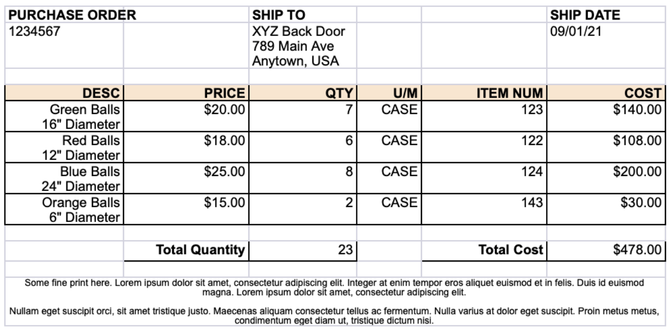
TOTAL QTY.
23

TOTAL COST
$478.00

Fig. 3. A templated scanned image for a purchase order.

Similarly, we might receive a scanned image of a purchase order that equally follows a template (like that in “Fig. 3”), but must be read using OCR. Processing the file through a basic OCR reader could provide us with a data structure (see snippets of the type of data an OCR engine can produce in Appendix **??**), which can then be organized through additional scripts, though a bit more cumbersome to handle. Assuming all scanned images received are provided in this same layout, we can still treat the documents received as structured data.

2) Semi-Structured Data: However, if the same layout template of the structured data isn’t followed, the changes or variations require new rules. New designs, layouts, or columns in an item table might be a result of different regulations for a region requiring additional table columns in a purchase order, or larger customers using different systems that generate the files being provided.

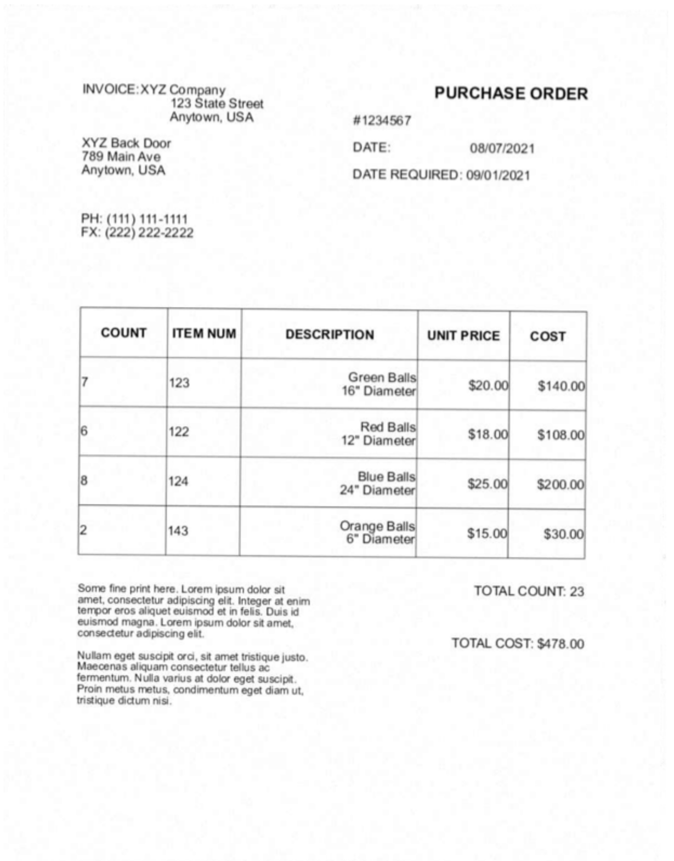


PURCHASE ORDER	**SHIP TO**				**SHIP DATE**
1234567	XYZ Back Door 789 Main Ave Anytown, USA				09/01/21
DESC	**PRICE**	**QTY**	**U/M**	**ITEM NUM**	**COST**
Green Balls 16" Diameter	$20.00	7	CASE	123	$140.00
Red Balls 12" Diameter	$18.00	6	CASE	122	$108.00
Blue Balls 24" Diameter	$25.00	8	CASE	124	$200.00
Orange Balls 6" Diameter	$15.00	2	CASE	143	$30.00
	Total Quantity	23		**Total Cost**	$478.00

Some fine print here. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Integer at enim tempor eros aliquet euismod et in felis. Duis id euismod magna. Lorem ipsum dolor sit amet, consectetur adipiscing elit.

Nullam eget suscipit orci, sit amet tristique justo. Maecenas aliquam consectetur tellus ac fermentum. Nulla varius at dolor eget suscipit. Proin metus metus, condimentum eget diam ut, tristique dictum nisi.

Fig. 4. A templated spreadsheet for a purchase order containing the same information as “Fig. 2”, but with a different layout.



INVOICE: XYZ Company
123 State Street
Anytown, USA

XYZ Back Door
789 Main Ave
Anytown, USA

PH: (111) 111-1111
FX: (222) 222-2222

PURCHASE ORDER

#1234567

DATE: 08/07/2021

DATE REQUIRED: 09/01/2021

COUNT	ITEM NUM	DESCRIPTION	UNIT PRICE	COST
7	123	Green Balls 16" Diameter	$20.00	$140.00
6	122	Red Balls 12" Diameter	$18.00	$108.00
8	124	Blue Balls 24" Diameter	$25.00	$200.00
2	143	Orange Balls 6" Diameter	$15.00	$30.00

Some fine print here. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Integer at enim tempor eros aliquet euismod et in felis. Duis id euismod magna. Lorem ipsum dolor sit amet, consectetur adipiscing elit.

Nullam eget suscipit orci, sit amet tristique justo. Maecenas aliquam consectetur tellus ac fermentum. Nulla varius at dolor eget suscipit. Proin metus metus, condimentum eget diam ut, tristique dictum nisi.

TOTAL COUNT: 23

TOTAL COST: $478.00

Fig. 5. A templated scanned image for a purchase order containing the same information as “Fig. 3”, but with a different layout.

Documents that contain the same type of information but in different designs and layouts fall into the category of semi-structured data. The keys in key-value pairs are the same across all documents, but the position on the page might vary, as seen in “Fig. 4”. Similarly, we might see scanned images like “Fig. 5” with varied layouts.

In this case, our rule-based approach gives us good accuracy for the document in “Fig. 2”, but low accuracy for the document in “Fig. 4” (and similarly with the scanned images). If a business process is limited to only a few variations in the layout, writing a unique rule for each type is manageable. Appendix C provides additional code that can be applied to the same methods used in Appendix A that will work for the spreadsheet in “Fig. 4”.

However, this approach will have difficulty scaling for large volumes of semi-structured data. Companies will have to create a template creation and management process, adding and adjusting scripts and rules with each change and new layout.

Introducing artificial intelligence (AI) at this stage for growing volumes of semi-structured data can make it easier to scale at this stage of document classification and data extraction, avoiding template management.

The field of AI combines computer science and hearty datasets in order to automate problem-solving [4]. Machine learning (ML), a specific form of AI, models can assist in automatically detecting information desired from the input documents. Vihar Kurama suggests the use of several pre-trained ML models (FastRCNN, Attention OCR, and Graph Convolution) for a hybrid solution of both rules-based data extraction and ML when working with semi-structured data such as our examples [5].

Important to note is that the reliance on automated extraction should always be kept in check by regular review of metrics, accuracy, and confidence scores. Thankfully the processing of more samples to a machine learning model results in higher accuracy of the data retrieved.

3) Unstructured Data: The final classification for documents received would be unstructured data. This type of document has no key-value pairs, may contain handwritten text, is free-flowing, and lacks consistency. Chances are that there are no tables (or the tables lack column headers) or a prescriptive way to automate the identification of which values to assign to which purposes.

Scripts and ML are not enough to understand and classify this data. Instead, we may need to consider other techniques, such as natural language processing (NLP). Another branch of artificial intelligence, NLP provides computers with the ability to understand the text of a document in the same way that human beings can [6].

C. Digitize & Standardize Data

With the process of understanding and classifying our data as structured, semi-structured, or unstructured, we now have some digital information that we can put to use. Our first pass at the documents might not have been deep enough to understand the document type; our business process may be more efficient at this point to gather just enough information to determine where to feed the document. This may mean that we’re capturing the fact that there is a purchase order number, or an invoice number, or the title of an application form.

In our business flow, we assume our employee may be receiving more than one document type and needs to determine what the document is before they can decide the appropriate actions to take. This same is true for an automated form of this process. By standardizing the reduced data we’ve collected, we can continue to feed the documents through a pipeline, knowing we can make a decision based on the key-value pairs we’ve found so far.

D. Understand Document Types

With the standardized data from the previous step, we should now be able to determine if the document is an invoice, a purchase order, an application, etc. Documents that contain purchase order numbers would get routed to the process for the ordering system, those with invoice numbers might be sent to the Accounts Payable team, and applications should be routed to the appropriate department as well.

As a human, understanding the document types only takes a glance. If we consider automating this step for our pipeline, we need enough information to make the decision. If our data is structured, a rules-based system makes this very easy. Semi-structured and unstructured data can make this more difficult, especially if many document types are ingested into the same system.

Depending on the complexity of the business process involved, we may consider introducing machine learning in this step. Or, we improve our ingestion step to separate out document types by directing users to submit different types of documents to the appropriate email address, shared folder, web portal, etc. The latter option may have some friction at the beginning but could be a very simple thing to overcome with a little customer training.

E. Better Data Extraction

Once we know our document type (invoice vs. purchase order, etc.) we can now apply more specific data extraction methods. This may be more heavily focused on machine learning models if the documents are semi-structured or unstructured.

Because we’ll assume that our data is semi-structured for our ball factory, this stage is a good placement for intelligent OCR. Model-based OCR engines can improve the process of digitizing the documents at scale while continuing to reduce errors with every training.

While many intelligent OCR tools are available, one such tool is the ABBYY FlexiCapture. In a review done by PricewaterhouseCoopers (PwC), ABBYY FlexiCapture was rated best in class in data validation and data classification, as well as data extraction in both structured and semi-structured documents. It also ranked well for multi-lingual capabilities, and the ability to read barcodes and tables. Google Tesseract and Microsoft’s Modi ranked better for screenscraping in desktop, web, and documents. PwC goes on to note that “the integration of the ABBYY OCR engine not only enhances automation for rules-driven processes, but also adds the flavour of NLP and widens the scope of automation." [7]

F. Human Validation

Nearing the end of our process flow, we've now reached the point where our data has been ingested and completely digitized into a standardized format. While it's true that with machine learning we may need fewer human checks as we train with more samples, a certain level of human-in-the-loop is necessary (and a good idea) when automating document handling. Confirming accurate information has been gathered from the ingested documents is vital for clean data and correct orders being created.

Thankfully, the continuous feed of more samples over time improves the ML model and builds confidence in the process. Regardless there should always be a human check to verify incoming information.

V. Single Pipeline

All of this has helped us understand the problem and an approach to handling documents. We have an idea of which types of tools might be useful and each particular step, but now we must understand how to string it all together into a single, cohesive pipeline.

A business process might begin as a simple macro or script, improving the time and effort needed for small tasks. From there, the process may see continued improvements through IT or business solutions. At some point, additional automation can be considered to raise the bar to another level: robotic process automation.

A. Robotic Process Automation

Robotic process automation, more commonly referred to as RPA, is the method of automating repetitive, often administrative, tasks. Generally speaking, virtual workers created for RPA will take advantage of a user interface on a screen, much like a human worker would do. These digital assistants will capture data and manipulate applications based on programmed decision-making. Similar to writing a programming script, RPA bots are made to mimic the same steps a human would do in particularly repetitive tasks, freeing up the human workers to focus on more valuable tasks.

While a powerful tool, the category of RPA is still relatively new in the technology field. In a survey of 100 document automation leaders by the Association of Intelligent Information Management, 18% of the respondents use RPA and 12% weren't sure of what RPA was. RPA with advanced capture has a place in the plans of 25% of the respondents, but 42% don't plan to make use of RPA. [8]

Beyond standard RPA, where we simply automate the flow itself, we can level up again by using intelligent RPA, which leverages forms of artificial intelligence.

The flow in "Fig. 6" is similar to the business process we began within "Fig. 1." However, we can see that with the use of RPA, we can assign many of previous stages to a virtual worker. With the rules and AI in place, our digital assistant

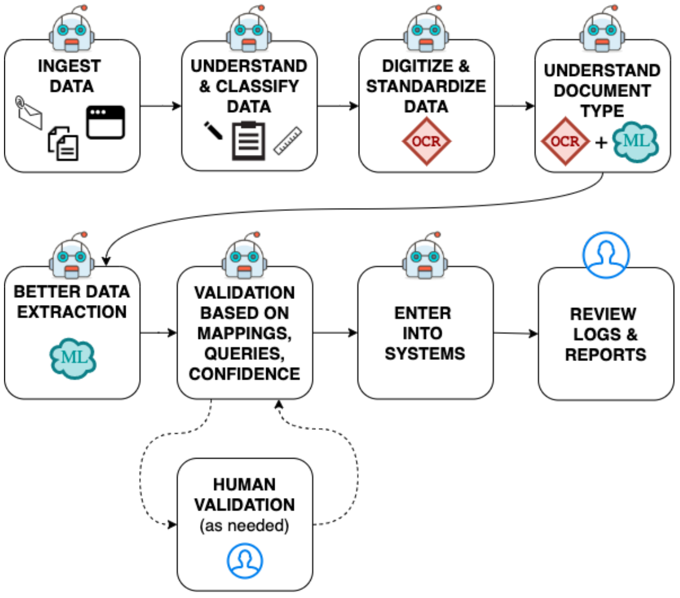

INGEST DATA
UNDERSTAND & CLASSIFY DATA
DIGITIZE & STANDARDIZE DATA
OCR
UNDERSTAND DOCUMENT TYPE
OCR + ML
BETTER DATA EXTRACTION
ML
VALIDATION BASED ON MAPPINGS, QUERIES, CONFIDENCE
ENTER INTO SYSTEMS
REVIEW LOGS & REPORTS
HUMAN VALIDATION (as needed)

Fig. 6. Document Flow in a Business Process with RPA Bot and Human Worker ownership for each step.

can not only ingest the data, but also classify, organize by type, standardize the data, validate it by cross-checking against existing table mappings (and alerting a human employee if further inspection is needed), but then also enter the data into the appropriate system. Beyond that, the system can be a legacy system that has no convenient integrations; RPA can interact with the user interface just like a human employee would, but faster and without error.

This intelligent enhancement to a previously cumbersome and very manual flow frees up the human employees to turn their focus back to what matters: the value of customer relationships and understanding their needs in order to grow sales.

B. Case Study: PepsiCo

PepsiCo faced the challenge of a document processing flow that was slow, prone to errors, and very manual. When they applied a single process flow and involved the ABBYY FlexiCapture solution to their documents, they found that they had lower error rates, reduced processing times, and their Accounts Payable staff were able to work much more efficiently [9].

PepsiCo's new flow is as follows:

1) Paper invoices (in any of 5 languages) are scanned locally
2) Scanned invoices are...
   a) sent to the POWER Project
   b) identified with the correct entity
   c) recognized by FlexiCapture
3) New digital documents and data are sent to verification stations
4) Data is verified

5) Data is forwarded to PepsiCo's Image Vision for approval
6) Data is exported in XML
7) Data is processed in PepsiCo's SAP ERP solution

"FlexiCapture's accuracy is also proving stable under a heavy workload. In its first three months the POWER Project solution has processed two thousand batches, over 21,000 documents and nearly 40,000 pages without issues – in a mix of five languages." [9]

This single pipeline solution has proven that it can scale and scale well! Powerful tools like ABBYY FlexiCapture combined with the power of RPA enabled PepsiCo to not only improve the intake process for documents at one location but to create a solution that accurately processes the information and can be scaled across the entire enterprise.

C. Case Study: Sysco

With a similar desire to improve the business processes, and in some cases, a need for automation due to fewer employees as a result of the COVID-19 pandemic, Sysco set out to also improve the accuracy and efficiency of a number of tasks across the company. By creating an RPA team, Sysco sought not to improve a single process, but to create a new culture around automation that could be applied enterprise-wide.

The Sysco RPA team has contributed to providing more than 200,000 hours of productivity given back to the business over the course of its first 3.5 years of existence [10].

While the team did create at least one automation that involved OCR tools used to parse digital faxes, the Sysco team explored a number of other solutions varying from simple rules-based automations to more complicated and impactful processes [11]. Interestingly, this team not only standardized information received for each of the processes that they automated but standardized their own process flow in order to improve their own efficiency and accuracy in bot creation.

TK: MORE HERE!

D. General Application

While the case studies presented above provide compelling narratives of the power of RPA in combination with OCR and AI, it may be difficult to understand where to begin. By identifying business cases that are repetitive and have a volume of effort that could be reduced through the use of scripts or automations, we have a place to start.

With a number of powerful solutions out there, benefits can be found in small steps first. Are employees typing in data from paper forms on a regular basis? Begin by ensuring that those documents are scanned into a system and then run an OCR engine against the document, providing your employees with something they can at least copy-paste. Or, perhaps you already have the information in structured spreadsheets. Write a script that automatically captures the key-value pairs and table data, storing the information in a standardized format like a database. Once the process flow is working smoothly to the point of consistent and standardized information, consider adding some type of automated solution that can enter the data directly into the system, rather than an employee copying the information in.

By taking small, iterative steps, any company, large or small, can work its way into an automated flow. While some solutions like ABBYY FlexiCapture can provide top-of-the-line solutions, the cost may be prohibitive to some companies. Start with the standard, rules-based methods, and enhance the areas that will still provide enough return-on-investment to justify the effort. The beautiful thing about the single-process pipeline idea is that more document types could be ingested over time, training the process with where to send each type. Digital worker steps might continue to be human employee steps, but there may be small improvements that can be made to allow the human to perform that step faster, like a dashboard that gives the employee a preview of the document, and they can quickly glance and classify the information manually. This may save on the implementation of intelligent OCR or ML while still providing time-saving benefits.

VI. Conclusion

Manual data entry is time-consuming and error-prone. The introduction of OCR (combined with ML) or NLP can result in fewer typos and opens the door to the improved process flow. Beyond that, considering RPA within the business process can create further efficiencies, giving skilled employees time back to focus on more valuable tasks that have a higher impact on the company's profitability.

A number of well-tested solutions are available, with pre-trained models available to implement from the start, reducing the number of required samples to get a machine learning solution off the ground quickly. However, the strongest solutions can be cost-prohibitive. Not to be deterred, we can still provide improvements by automating smaller sections of the process flow or enhancing the human step (thus bypassing the need for intelligent automation) in order to make it simpler to perform.

While still a young category of technology, RPA provides further enhancements by not only stringing together the steps into a single process flow but also moving all the way into working with legacy systems without the need to wait for modern integrations like APIs.

Depending on the business needs, more research can (and should) be done on which tools may provide the best solutions. If the budget allows, ABBYY FlexiCapture was rated very highly. A quick search on the web can provide appropriate overviews of the best tools. For example, Nanonets provides a list of pros and cons for the best OCR tools of 2021 [12]. Depending on the needs and time available (for development and potential training of models) for each business case, the right tool for the job may change.

References

[1] Matt Harris. When good info goes bad: The real cost of human data errors, 2014. [Online; accessed 15-August2021].

[2] James Schneider Ph.D., Bill Schultz, Julia McCrimlisk, Jesse Hulsing, and Ryan M. Nash CFA. B2B: How the next payments frontier will unleash small business. EQUITY RESEARCH, 2018. [Online; accessed 13-August2021].

[3] Justin Lavelle. Gartner says robotic process automation can save finance departments 25,000 hours of avoidable work annually, 2019. [Online; accessed 13-August2021].

[4] IBM Cloud Education. Artificial intelligence (AI), 2020. [Online; accessed 15-August2021].

[5] Vihar Kurama. A comprehensive guide to OCR with RPA and document understanding, 2021. [Online; accessed 13-August2021].

[6] IBM Cloud Education. Natural language processing (NLP), 2020. [Online; accessed 14-August2021].

[7] PwC. Robotic process automation and intelligent character recognition: Smart data capture. PricewaterhouseCoopers Private Limited, 2018. [Online; accessed 13-August2021].

[8] Garrett Hollander. [survey results] intelligent document automation trends, 2019. [Online; accessed 13-August2021].

[9] PepsiCo automates invoice processing with ABBYY FlexiCapture, 2021. [Online; accessed 13-August2021].

[10] Alison Major and Kim Meredith. Sysco: Scaling an rpa program without compromise, 2021. [Online; accessed 15-August2021].

[11] Blue Prism Café. Sysco: Scaling an intelligent automation program without compromise, 2021. [Webinar; accessed 15-August2021].

[12] Prithiv S. Best ocr software of 2021, 2021. [Online; accessed 15-August2021].

Appendix A
Python Methods to Collect Data from a Spreadsheet

A. *spreadsheets/methods.py*

```python
import string


def get_po_number(workbook, cells):
    """Returns the value for the Purchase Order Number as a string"""
    value = get_cell_value(workbook, cells["poNum"])
    value_as_int = int(value)
    value_as_string = str(value_as_int)
    return value_as_string


def get_bill_to_name(workbook, cells):
    """Returns the value for the Bill-To Name"""
    return get_cell_value(workbook, cells["billTo"])


def get_ship_to_name(workbook, cells):
    """Returns the value for the Ship-To Name (only first line)"""
    value = get_cell_value(workbook, cells["shipTo"])
    value_after_first_line = value.split("\n")
    return value_after_first_line[0]


def get_ship_date(workbook, cells):
    """Returns the value for the Shipping Date"""
    return get_cell_value(workbook, cells["shipDate"])


def get_total_quantity(workbook, cells):
    """Returns the value for the Total Quantity as an integer"""
    value = get_cell_value(workbook, cells["totalQty"])
    value_as_int = int(value)
    return value_as_int


def get_total_cost(workbook, cells):
    """Returns the value for the Total Cost"""
    return get_cell_value(workbook, cells["totalCost"])


def get_cell_value(workbook, cell):
    """Gets the value of the cell (ex: 'B1') from a workbook"""
    if cell == "":
        """No cell was provided."""
        return "[cell not mapped]"

    column = get_column_value(cell)
    row = get_row_value(cell)
    return workbook.loc[row].at[column]


def get_column_value(cell):
    """Gets the 'loc' value from a cell"""
    alphabet = dict()
    for index, letter in enumerate(string.ascii_lowercase):
        alphabet[letter] = index

    value = cell[0].lower()
    return alphabet[value]


def get_row_value(cell):
    """Gets the 'at' value from a cell"""
    value = cell[1:]
    return int(value) - 1


def get_item_line_number(item_table, item_row):
    """Gets the item's line number from the table"""
    try:
        return item_table.at[item_row, 'LINE_NUM']
    except KeyError:
        return "[not available]"


def get_item_quantity(item_table, item_row):
    """Gets the item's quantity from the table"""
    try:
        return item_table.at[item_row, 'QTY']
    except KeyError:
        return "[not available]"


def get_item_unit_of_measure(item_table, item_row):
    """Gets the item's unit of measure from the table"""
    try:
        return item_table.at[item_row, 'UOM']
    except KeyError:
        return "[not available]"


def get_item_number(item_table, item_row):
    """Gets the item's identification number from the table"""
    try:
        value = item_table.at[item_row, 'ITEM_NUM']
        value_as_int = int(value)
        return str(value_as_int)
    except KeyError:
        return "[not available]"


def get_item_description(item_table, item_row):
    """Gets the item's description from the table"""
    try:
        return item_table.at[item_row, 'DESC']
    except KeyError:
        return "[not available]"


def get_item_unit_price(item_table, item_row):
    """Gets the item's unit price from the table"""
    try:
        return item_table.at[item_row, 'UNIT_PRICE']
    except KeyError:
        return "[not available]"


def get_item_cost(item_table, item_row):
    """Gets the item's total order cost from the table"""
    try:
        return item_table.at[item_row, 'COST']
    except KeyError:
        return "[not available]"
```

B. spreadsheets/excel_order_one.py

These mappings are specific to the spreadsheet represented in “Fig. 2”.

```python
import pandas as pd

FILE_PATH = '../documents/ORDERS.xlsx'
SHEET_NAME = 'Order Sample 1'

"""The workbook to pull values from"""
WORKBOOK = pd.read_excel(
    FILE_PATH,
    sheet_name=SHEET_NAME,
    header=None
)

"""The table for item information"""
ITEM_TABLE = pd.read_excel(
    FILE_PATH,
    sheet_name=SHEET_NAME,
    usecols="A:G",
    skiprows=5,
    nrows=4
)
"""Mapping column names for item table"""
ITEM_TABLE.columns = [
    'LINE_NUM',     # Line #
    'QTY',          # Quantity
    'UOM',          # Unit of Measure
    'ITEM_NUM',     # Item #
    'DESC',         # Description
    'UNIT_PRICE',   # Unit Price
    'COST'          # Charge
]

"""Mapping of the values to cells"""
ORDER_ONE_CELLS = {
    "poNum": "B1",
    "billTo": "B4",
    "shipTo": "B3",
    "shipDate": "B2",
    "totalQty": "B12",
    "totalCost": "G12"
}
```

APPENDIX B
UNIT TESTS FOR PYTHON METHODS IN APPENDIX A

A. *tests/test_spreadsheets_methods.py*

```python
import datetime
import unittest
import pandas as pd
from parameterized import parameterized
from spreadsheets.methods import get_column_value, get_row_value, get_po_number, get_total_cost, get_to
    get_ship_date, get_ship_to_name, get_bill_to_name, get_cell_value, get_item_line_number, get_item_q
    get_item_unit_of_measure, get_item_number, get_item_description, get_item_unit_price, get_item_cost

FILE_PATH = '../documents/ORDERS.xlsx'
SHEET_NAME = 'Order Sample 1'

"""The workbook to pull values from"""
WORKBOOK = pd.read_excel(
    FILE_PATH,
    sheet_name=SHEET_NAME,
    header=None
)

"""Mapping of the values to cells"""
CELLS = {
    "poNum": "B1",
    "billTo": "B4",
    "shipTo": "B3",
    "shipDate": "B2",
    "totalQty": "B12",
    "totalCost": "G12"
}

"""Fake item table for testing"""
ITEM_TABLE = pd.DataFrame(
    [
        [1, 7, 'CASE', '123', 'Item 1', 20, 140],
        [2, 6, 'PALLET', 124.0, 'Item 2', 18, 108]
    ],
    columns=[
        'LINE_NUM',
        'QTY',
        'UOM',
        'ITEM_NUM',
        'DESC',
        'UNIT_PRICE',
        'COST'
    ]
)


class SpreadsheetsMethods(unittest.TestCase):
    @parameterized.expand([
        ["B1 gives 1", "B1", 1],
        ["B2 gives 1", "B2", 1],
        ["A1 gives 0", "A1", 0],
        ["Z9 gives 25", "Z9", 25]
    ])
    def test_get_column_value(self, name, cell, expected):
        actual = get_column_value(cell)
        self.assertEqual(expected, actual)

    @parameterized.expand([
        ["B1 gives 1", "B1", 0],
        ["B2 gives 1", "B2", 1],
        ["A1 gives 0", "A1", 0],
        ["Z9 gives 25", "Z9", 8],
        ["F13 gives 12", "F13", 12],
        ["D101 gives 100", "D101", 100]
    ])
    def test_get_row_value(self, name, cell, expected):
        actual = get_row_value(cell)
        self.assertEqual(expected, actual)

    @parameterized.expand([
        ["A1 gives 'PO#'", "A1", "PO#"],
        ["blank gives '[cell not mapped]'", "", "[cell not mapped]"]
    ])
    def test_get_cell_value(self, name, cell, expected):
        actual = get_cell_value(WORKBOOK, cell)
        self.assertEqual(expected, actual)

    def test_get_po_number(self):
        expected = "1234567"
        actual = get_po_number(WORKBOOK, CELLS)
        self.assertEqual(expected, actual)

    def test_get_bill_to_name(self):
        expected = "XYZ Company"
        actual = get_bill_to_name(WORKBOOK, CELLS)
        self.assertEqual(expected, actual)

    def test_get_ship_to_name(self):
        expected = "XYZ Back Door"
        actual = get_ship_to_name(WORKBOOK, CELLS)
        self.assertEqual(expected, actual)

    def test_get_ship_date(self):
        expected = datetime.datetime(2021, 9, 1, 0, 0)  # 09/01/21
        actual = get_ship_date(WORKBOOK, CELLS)
        self.assertEqual(expected, actual)

    def test_get_total_quantity(self):
        expected = 23
        actual = get_total_quantity(WORKBOOK, CELLS)
        self.assertEqual(expected, actual)

    def test_get_total_cost(self):
        expected = 478.00
        actual = get_total_cost(WORKBOOK, CELLS)
        self.assertEqual(expected, actual)

    @parameterized.expand([
        ["Row 1 gives line '1'", ITEM_TABLE, 0, 1],
        ["Row 2 gives line '2'", ITEM_TABLE, 1, 2],
        ["Row 3 gives '[not available]'", ITEM_TABLE, 2, '[not available]']
    ])
    def test_get_item_line_number(self, name, table, row, expected):
        actual = get_item_line_number(table, row)
        self.assertEqual(expected, actual)

    @parameterized.expand([
        ["Row 1 gives quantity 7", ITEM_TABLE, 0, 7],
        ["Row 2 gives quantity 6", ITEM_TABLE, 1, 6],
        ["Row 3 gives '[not available]'", ITEM_TABLE, 2, '[not available]']
    ])
    def test_get_item_quantity(self, name, table, row, expected):
        actual = get_item_quantity(table, row)
        self.assertEqual(expected, actual)

    @parameterized.expand([
        ["Row 1 gives 'CASE'", ITEM_TABLE, 0, 'CASE'],
        ["Row 2 gives 'PALLET'", ITEM_TABLE, 1, 'PALLET'],
        ["Row 3 gives '[not available]'", ITEM_TABLE, 2, '[not available]']
    ])
    def test_get_item_unit_of_measure(self, name, table, row, expected):
        actual = get_item_unit_of_measure(table, row)
        self.assertEqual(expected, actual)

    @parameterized.expand([
        ["Row 1 gives item number '123'", ITEM_TABLE, 0, '123'],
        ["Row 2 gives item number '124'", ITEM_TABLE, 1, '124'],
        ["Row 3 gives '[not available]'", ITEM_TABLE, 2, '[not available]']
    ])
    def test_get_item_number(self, name, table, row, expected):
        actual = get_item_number(table, row)
        self.assertEqual(expected, actual)

    @parameterized.expand([
        ["Row 1 gives 'Item 1'", ITEM_TABLE, 0, 'Item 1'],
        ["Row 2 gives 'Item 2'", ITEM_TABLE, 1, 'Item 2'],
        ["Row 3 gives '[not available]'", ITEM_TABLE, 2, '[not available]']
    ])
    def test_get_item_description(self, name, table, row, expected):
        actual = get_item_description(table, row)
        self.assertEqual(expected, actual)

    @parameterized.expand([
        ["Row 1 gives 20.00", ITEM_TABLE, 0, 20.00],
        ["Row 2 gives 18.00", ITEM_TABLE, 1, 18.00],
        ["Row 3 gives '[not available]'", ITEM_TABLE, 2, '[not available]']
    ])
    def test_get_item_unit_price(self, name, table, row, expected):
        actual = get_item_unit_price(table, row)
        self.assertEqual(expected, actual)

    @parameterized.expand([
        ["Row 1 gives 140.00", ITEM_TABLE, 0, 140.00],
        ["Row 2 gives 108.00", ITEM_TABLE, 1, 108.00],
        ["Row 3 gives '[not available]'", ITEM_TABLE, 2, '[not available]']
    ])
    def test_get_item_cost(self, name, table, row, expected):
        actual = get_item_cost(table, row)
        self.assertEqual(expected, actual)


if __name__ == '__main__':
    unittest.main()
```

B. *tests/test_excel_order_one.py*

```python
import datetime
import unittest
from parameterized import parameterized
from spreadsheets.excel_order_one import WORKBOOK, ORDER_ONE_CELLS, ITEM_TABLE
from spreadsheets.methods import get_po_number, get_bill_to_name, get_ship_to_name, get_ship_date, get_
    get_total_cost, get_item_line_number, get_item_quantity, get_item_unit_of_measure, get_item_number,
    get_item_description, get_item_unit_price, get_item_cost


class ExcelOrderOne(unittest.TestCase):
    def test_get_po_number(self):
        expected = "1234567"
        actual = get_po_number(WORKBOOK, ORDER_ONE_CELLS)
        self.assertEqual(expected, actual)

    def test_get_bill_to_name(self):
        expected = "XYZ Company"
        actual = get_bill_to_name(WORKBOOK, ORDER_ONE_CELLS)
        self.assertEqual(expected, actual)

    def test_get_ship_to_name(self):
        expected = "XYZ Back Door"
        actual = get_ship_to_name(WORKBOOK, ORDER_ONE_CELLS)
        self.assertEqual(expected, actual)

    def test_get_ship_date(self):
        expected = datetime.datetime(2021, 9, 1, 0, 0)  # 09/01/21
        actual = get_ship_date(WORKBOOK, ORDER_ONE_CELLS)
        self.assertEqual(expected, actual)

    def test_get_total_quantity(self):
        expected = 23
        actual = get_total_quantity(WORKBOOK, ORDER_ONE_CELLS)
        self.assertEqual(expected, actual)

    def test_get_total_cost(self):
        expected = 478.00
        actual = get_total_cost(WORKBOOK, ORDER_ONE_CELLS)
        self.assertEqual(expected, actual)

    @parameterized.expand([
        ["Row 1 gives line '1'", ITEM_TABLE, 0, 1],
        ["Row 2 gives line '2'", ITEM_TABLE, 1, 2],
        ["Row 3 gives line '3'", ITEM_TABLE, 2, 3],
        ["Row 4 gives line '4'", ITEM_TABLE, 3, 4]
    ])
    def test_get_item_line_number(self, name, table, row, expected):
        actual = get_item_line_number(table, row)
        self.assertEqual(expected, actual)

    @parameterized.expand([
        ["Row 1 gives quantity 7", ITEM_TABLE, 0, 7],
        ["Row 2 gives quantity 6", ITEM_TABLE, 1, 6],
        ["Row 3 gives quantity 8", ITEM_TABLE, 2, 8],
        ["Row 4 gives quantity 2", ITEM_TABLE, 3, 2]
    ])
    def test_get_item_quantity(self, name, table, row, expected):
        actual = get_item_quantity(table, row)
        self.assertEqual(expected, actual)

    @parameterized.expand([
        ["Row 1 gives 'CASE'", ITEM_TABLE, 0, 'CASE'],
        ["Row 2 gives 'CASE'", ITEM_TABLE, 1, 'CASE'],
        ["Row 3 gives 'CASE'", ITEM_TABLE, 2, 'CASE'],
        ["Row 4 gives 'CASE'", ITEM_TABLE, 3, 'CASE']
    ])
    def test_get_item_unit_of_measure(self, name, table, row, expected):
        actual = get_item_unit_of_measure(table, row)
        self.assertEqual(expected, actual)

    @parameterized.expand([
        ["Row 1 gives item number '123'", ITEM_TABLE, 0, '123'],
        ["Row 2 gives item number '124'", ITEM_TABLE, 1, '122'],
        ["Row 3 gives item number '123'", ITEM_TABLE, 2, '124'],
        ["Row 4 gives item number '123'", ITEM_TABLE, 3, '143']
    ])
    def test_get_item_number(self, name, table, row, expected):
        actual = get_item_number(table, row)
        self.assertEqual(expected, actual)

    @parameterized.expand([
        ["Row 1 gives the description", ITEM_TABLE, 0, 'Green Balls\n16" Diameter'],
        ["Row 2 gives the description", ITEM_TABLE, 1, 'Red Balls\n12" Diameter'],
        ["Row 3 gives the description", ITEM_TABLE, 2, 'Blue Balls\n24" Diameter'],
        ["Row 4 gives the description", ITEM_TABLE, 3, 'Orange Balls\n6" Diameter']
    ])
    def test_get_item_description(self, name, table, row, expected):
        actual = get_item_description(table, row)
        self.assertEqual(expected, actual)

    @parameterized.expand([
        ["Row 1 gives 20.00", ITEM_TABLE, 0, 20.00],
        ["Row 2 gives 18.00", ITEM_TABLE, 1, 18.00],
        ["Row 3 gives 25.00", ITEM_TABLE, 2, 25.00],
        ["Row 4 gives 15.00", ITEM_TABLE, 3, 15.00]
    ])
    def test_get_item_unit_price(self, name, table, row, expected):
        actual = get_item_unit_price(table, row)
        self.assertEqual(expected, actual)

    @parameterized.expand([
        ["Row 1 gives 140.00", ITEM_TABLE, 0, 140.00],
        ["Row 2 gives 108.00", ITEM_TABLE, 1, 108.00],
        ["Row 3 gives 200.00", ITEM_TABLE, 2, 200.00],
        ["Row 4 gives 30.00", ITEM_TABLE, 3, 30.00]
    ])
    def test_get_item_cost(self, name, table, row, expected):
        actual = get_item_cost(table, row)
        self.assertEqual(expected, actual)


if __name__ == '__main__':
    unittest.main()
```

Appendix C
Python Methods to Collect Data from a Spreadsheet

A. *spreadsheets/excel_order_two.py*

These mappings are specific to the spreadsheet represented in "Fig. 4".

```python
import pandas as pd

FILE_PATH = '../documents/ORDERS.xlsx'
SHEET_NAME = 'Order Sample 2'

"""The workbook to pull values from"""
WORKBOOK = pd.read_excel(
    FILE_PATH,
    sheet_name=SHEET_NAME,
    header=None
)

"""The table for item information"""
ITEM_TABLE = pd.read_excel(
    FILE_PATH,
    sheet_name=SHEET_NAME,
    usecols="A:F",
    skiprows=3,
    nrows=4
)
"""Mapping column names for item table"""
ITEM_TABLE.columns = [
    'DESC',         # DESC
    'UNIT_PRICE',   # PRICE
    'QTY',          # QTY
    'UOM',          # U/M
    'ITEM_NUM',     # ITEM_NUM
    'COST'          # COST
]

"""Mapping of the values to cells"""
ORDER_TWO_CELLS = {
    "poNum": "A2",
    "billTo": "",
    "shipTo": "C2",
    "shipDate": "F2",
    "totalQty": "C10",
    "totalCost": "F10"
}
```

Appendix D
Unit Tests for Python Mappings for Spreadsheet Order Two

A. *tests/test_excel_order_two.py*

```python
import datetime
import unittest
from parameterized import parameterized
from spreadsheets.excel_order_two import WORKBOOK, ORDER_TWO_CELLS, ITEM_TABLE
from spreadsheets.methods import get_po_number, get_bill_to_name, get_ship_to_name, get_ship_date, get_
    get_total_cost, get_item_line_number, get_item_quantity, get_item_unit_of_measure, get_item_number,
    get_item_description, get_item_unit_price, get_item_cost


class ExcelOrderTwo(unittest.TestCase):
    def test_get_po_number(self):
        expected = "1234567"
        actual = get_po_number(WORKBOOK, ORDER_TWO_CELLS)
        self.assertEqual(expected, actual)

    def test_get_bill_to_name(self):
        expected = "[cell not mapped]"
        actual = get_bill_to_name(WORKBOOK, ORDER_TWO_CELLS)
        self.assertEqual(expected, actual)

    def test_get_ship_to_name(self):
        expected = "XYZ Back Door"
        actual = get_ship_to_name(WORKBOOK, ORDER_TWO_CELLS)
        self.assertEqual(expected, actual)

    def test_get_ship_date(self):
        expected = datetime.datetime(2021, 9, 1, 0, 0)  # 09/01/21
        actual = get_ship_date(WORKBOOK, ORDER_TWO_CELLS)
        self.assertEqual(expected, actual)

    def test_get_total_quantity(self):
        expected = 23
        actual = get_total_quantity(WORKBOOK, ORDER_TWO_CELLS)
        self.assertEqual(expected, actual)

    def test_get_total_cost(self):
        expected = 478.00
        actual = get_total_cost(WORKBOOK, ORDER_TWO_CELLS)
        self.assertEqual(expected, actual)

    @parameterized.expand([
        ["Row 1 gives [not available]", ITEM_TABLE, 0, "[not available]"],
        ["Row 2 gives [not available]", ITEM_TABLE, 1, "[not available]"],
        ["Row 3 gives [not available]", ITEM_TABLE, 2, "[not available]"],
        ["Row 4 gives [not available]", ITEM_TABLE, 3, "[not available]"]
    ])
    def test_get_item_line_number(self, name, table, row, expected):
        actual = get_item_line_number(table, row)
        self.assertEqual(expected, actual)

    @parameterized.expand([
        ["Row 1 gives quantity 7", ITEM_TABLE, 0, 7],
        ["Row 2 gives quantity 6", ITEM_TABLE, 1, 6],
        ["Row 3 gives quantity 8", ITEM_TABLE, 2, 8],
        ["Row 4 gives quantity 2", ITEM_TABLE, 3, 2]
    ])
    def test_get_item_quantity(self, name, table, row, expected):
        actual = get_item_quantity(table, row)
        self.assertEqual(expected, actual)

    @parameterized.expand([
        ["Row 1 gives 'CASE'", ITEM_TABLE, 0, 'CASE'],
        ["Row 2 gives 'CASE'", ITEM_TABLE, 1, 'CASE'],
        ["Row 3 gives 'CASE'", ITEM_TABLE, 2, 'CASE'],
        ["Row 4 gives 'CASE'", ITEM_TABLE, 3, 'CASE']
    ])
    def test_get_item_unit_of_measure(self, name, table, row, expected):
        actual = get_item_unit_of_measure(table, row)
        self.assertEqual(expected, actual)

    @parameterized.expand([
        ["Row 1 gives item number '123'", ITEM_TABLE, 0, '123'],
        ["Row 2 gives item number '124'", ITEM_TABLE, 1, '122'],
        ["Row 3 gives item number '123'", ITEM_TABLE, 2, '124'],
        ["Row 4 gives item number '123'", ITEM_TABLE, 3, '143']
    ])
    def test_get_item_number(self, name, table, row, expected):
        actual = get_item_number(table, row)
        self.assertEqual(expected, actual)

    @parameterized.expand([
        ["Row 1 gives the description", ITEM_TABLE, 0, 'Green Balls\n16" Diameter'],
        ["Row 2 gives the description", ITEM_TABLE, 1, 'Red Balls\n12" Diameter'],
        ["Row 3 gives the description", ITEM_TABLE, 2, 'Blue Balls\n24" Diameter'],
        ["Row 4 gives the description", ITEM_TABLE, 3, 'Orange Balls\n6" Diameter']
    ])
    def test_get_item_description(self, name, table, row, expected):
        actual = get_item_description(table, row)
        self.assertEqual(expected, actual)

    @parameterized.expand([
        ["Row 1 gives 20.00", ITEM_TABLE, 0, 20.00],
        ["Row 2 gives 18.00", ITEM_TABLE, 1, 18.00],
        ["Row 3 gives 25.00", ITEM_TABLE, 2, 25.00],
        ["Row 4 gives 15.00", ITEM_TABLE, 3, 15.00]
    ])
    def test_get_item_unit_price(self, name, table, row, expected):
        actual = get_item_unit_price(table, row)
        self.assertEqual(expected, actual)

    @parameterized.expand([
        ["Row 1 gives 140.00", ITEM_TABLE, 0, 140.00],
        ["Row 2 gives 108.00", ITEM_TABLE, 1, 108.00],
        ["Row 3 gives 200.00", ITEM_TABLE, 2, 200.00],
        ["Row 4 gives 30.00", ITEM_TABLE, 3, 30.00]
    ])
    def test_get_item_cost(self, name, table, row, expected):
        actual = get_item_cost(table, row)
        self.assertEqual(expected, actual)


if __name__ == '__main__':
    unittest.main()
```