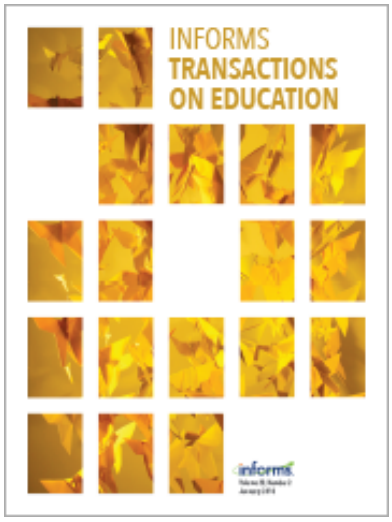


INFORMS Transactions on Education

Publication details, including instructions for authors and subscription information:
http://pubsonline.informs.org

Spreadsheet Modeling and Wrangling with Python

Mark W. Isken

To cite this article:
Mark W. Isken (2025) Spreadsheet Modeling and Wrangling with Python. INFORMS Transactions on Education 25(2):152-168. https://doi.org/10.1287/ited.2023.0047

https://pubsonline.informs.org/journal/ited

INFORMS TRANSACTIONS ON EDUCATION
Vol. 25, No. 2, January 2025, pp. 152–168
ISSN 1532-0545 (online)

Spreadsheet Modeling and Wrangling with Python

Mark W. Isken[a]

[a] Decision and Information Sciences, School of Business Administration, Oakland University, Rochester, Michigan 48309
Contact: isken@oakland.edu, https://orcid.org/0000-0001-8471-9116 (MWI)

Received: July 28, 2023
Revised: November 21, 2023; December 5, 2023; February 6, 2024
Accepted: February 7, 2024
Published Online in Articles in Advance: March 15, 2024

https://doi.org/10.1287/ited.2023.0047

Abstract. A staple of many spreadsheet-based management science courses is the use of Excel for activities such as model building, sensitivity analysis, goal seeking, and Monte-Carlo simulation. What might those things look like if carried out using Python? We describe a teaching module in which Python is used to do typical Excel-based modeling and data-wrangling tasks. In addition, students are exposed to basic software engineering principles, including project folder structures, version control, object-oriented programming, and other more advanced Python skills, creating deployable packages and documentation. The module is supported with Jupyter notebooks, Python scripts, course web pages that include numerous screencasts, and a few GitHub repositories. All of the supporting materials are permissively licensed and freely accessible.

Supplemental Material: The supplemental files are available at https://doi.org/10.1287/ited.2023.0047.

Keywords: python • spreadsheet modeling

1. Introduction

Spreadsheet-based modeling has transformed the teaching of management science in business schools over the past 20 plus years. The late 1990s saw first editions of game-changing textbooks (Ragsdale 2017, Winston and Albright 2018) focused on the use of spreadsheets in teaching modeling and management science. A series of conferences focused on teaching management science with spreadsheets started at Dartmouth in 1998, and several presentations from that conference made up the bulk of papers in the first two issues of the *INFORMS Transactions on Education* (*ITE*) journal (Baker 2000, Bell 2000, Carraway and Clyman 2000, Evans 2000, Ragsdale 2001, Savage 2001). The next few decades saw numerous textbooks, including Powell and Baker (2019) and Camm et al. (2020), and teaching papers in journals, such as *ITE*, that prominently featured spreadsheet-based modeling.

Despite numerous calls for, and predictions of, the eradication of spreadsheets from the business world, Excel and other spreadsheets are alive and well. Their flexibility is unparalleled, and it is unrealistic to expect them to disappear anytime soon. If anything, the recent announcement by Microsoft that users will be able combine Python scripts with Excel formulas within the same workbook will likely solidify Excel's place in the analytics space. Yes, spreadsheets have well-known limitations. Spreadsheets are not databases and probably should not be used as one, especially for relational data models. There are better data visualization and exploration tools. There are better dashboarding tools that are more easily maintained in a production environment. Spreadsheet errors can be a huge problem. Version control is difficult as both data and logic live in the same spreadsheet document. Documentation of multistep manual data manipulation processes is often lacking—hindering reproducibility and automation. There are data size limitations, and external data links present their own set of challenges. However, Excel continues to be used for a wide range of quantitative analysis-related activities throughout the business world. One particular use is the building and exercising of models of business problems to explore the impact of important model inputs on different output metrics. This "what if?"-type modeling is facilitated by Excel's flexibility, numerous built-in functions, integrated sensitivity analysis via Data Tables, Goal Seek for break-even analysis and exploratory data analysis (EDA) via plots, and Pivot Tables.

At the same time, the analytics world has been changing. The advent of programmatic analytics using tools, like R and Python, has changed the practice of analytics as well as the teaching landscape. Limitations of spreadsheets with respect to reproducibility and automation have certainly played a role. Additionally, both R and Python are free and open source, and they

have a rich set of analytics-related libraries that have contributed to their growing popularity. There is a widely used Python distribution, Anaconda, that specifically targets analytics and data science work. Python, being a general purpose programming language, also offers advantages when moving models and analytical analysis pipelines from the prototype stage into production. Enormous data science ecosystems have emerged around both R and Python, providing a wealth of learning resources for both new and experienced analytics students and professionals.

Numerous R- and Python-based data science courses have been developed and offered at universities and through online learning platforms, such as Coursera, EdX, DataCamp, Udemy, and many others. One particularly relevant example is the courses in computation offered by the Naval Postgraduate School and described in Alderson (2022). In their courses, Jupyter notebooks (discussed in more detail in the next section) play a prominent role in providing an interactive development and learning environment for Python within an operations research context. In Babier et al. (2023), an analytics project-based course is described in which Python plays a significant role. Students use Python for a number of data-wrangling, exploratory data analysis, and modeling tasks. Through a course like this, students gain valuable experience with a tool that is becoming more of an expectation in the analytics world. Another effective and fun way that students can gain experience with Python is through using it solve puzzles, such as Wordle (Lakhani et al. 2023) or the Advent of Code (Wastl 2023).

In 2015, at Oakland University, we created a course called Practical Computing for Data Analytics (PCDA) that introduced both R and Python (along with a little Linux) for use in data analytics work. This new course was meant as a follow-up to the spreadsheet-based modeling course we have had in place since 2001. The PCDA course, like many introductory data science courses, focuses mostly on data wrangling, EDA, and statistical/machine learning-based predictive models. In teaching EDA using tools like ggplot2, students are challenged to think about how they would go about trying to use Excel to create some of the plots that are easy to do with R- and Python-based plotting libraries. There is some limited coverage of using Python to open Excel files and do worksheet manipulation. However, other than a Python-based simulation of the Monte-Hall three-door problem, there was no real attempt to do the kind of modeling that is the focus of most spreadsheet-based modeling textbooks.

Python has continued to make inroads to analytical territory that was formerly dominated by spreadsheets. The widely used Python data analysis library, pandas, was created by Wes McKinney while working as a quantitative financial analyst (McKinney 2022). Python is widely used for data wrangling, visualization, and predictive modeling using various flavors of regression, tree-based models, and neural networks. You can get a very good sense of this by visiting the very popular r/datascience subreddit on Reddit—a well-known online forum. However, it got me to thinking. What about those things for which Excel is particularly well suited, such as building formula-based models and doing sensitivity analysis on these models? What would those look like in Python? The importance of modeling, articulated so well by Powell (2001) in the pages of this journal, has not abated. What role does Python have to play in supporting the kind of modeling taught for years in our business schools?

In 2021, I published a series of blog posts that explored this question and formed the basis of the teaching module described in this paper. During the development of these posts, I came across the Quant Econ open source code for economic modeling project, which "is a nonprofit organization dedicated to development and documentation of open source computational tools for economics, econometrics, and decision making." This is a mature project, run by renowned economists, with significant support. They have developed and released numerous high-quality Jupyter notebooks exploring a range of computational economic and financial modeling topics. Here was an example of the Python for business usage I had been seeking. With this as inspiration, I decided to turn my blog post series into a teaching module called Excel with Python (EwP).

The EwP module is not as polished nor as advanced as the Quant Econ materials, but it does add to the growing body of freely available teaching materials aimed at Python for business modeling. My goals for EwP are also a little different than those of Quant Econ and include

- teaching relevant and more advanced Python to business students using familiar and simple modeling examples,
- conveying a sense of the software design and development process,
- acting as a tour guide for a learning journey, and
- teaching students to think and act like a software developer, not just an analyst.

Recently, there has been more recognition of the need for data scientists to learn basic software engineering principes (Rodrigues 2023, Treadway 2023, Nelson 2024). The image of newly minted data science graduates developing spaghetti-coded Jupyter notebooks not nearly ready to be put into production has reached meme-worthy status. I hoped to address some of these issues in the EwP module. Students would be exposed to some simple software engineering concepts during the process of building and analyzing familiar models. This is really no different than the integration

of advanced Excel functions and techniques with spreadsheet modeling tutorials. Just as spreadsheet modeling students were able to easily wow their coworkers and managers with INDEX-MATCH and intermediate Pivot Table skills, I hoped that my students' ability to properly document a Python function and use git and GitHub would give them a bit more professional credibility.

We will begin with a brief discussion of the overall positioning and structure of the EwP module within our business analytics (BA) programs at Oakland University. Then, each of the three main EwP submodules is described in some detail. Finally, we conclude with a brief discussion of our experience in delivering this module. All of the Jupyter notebooks, along with html versions of them, can be found in the Supplementary Materials.

2. Module Positioning and Structure

At our institution, the EwP module is part of a business course entitled Advanced Analytics with Python (AAP). The students taking this course have already taken the PCDA course where they learned fundamental Python programming within the context of data analytics over seven weeks. The PCDA course ends with an introduction to the scikit-learn library for doing predictive modeling in Python, and the AAP course begins where the PCDA course ends. The EwP module follows immediately after the modules on machine learning and builds on the Python fundamentals that were covered in the PCDA course. The EwP module could also be used as part of a semester-long Python-based analytics course, in which the Python modules covered in the PCDA course are followed by more advanced topics, such as EwP or other topics in our AAP course. The only reason the EwP module is not included in the PCDA course is that Python is only covered in half of that course—the other half being an introduction to R and Linux. If you visit the open-access PCDA course website and AAP course website, you can see the topic outline for each course and how the EwP module is positioned.

Most AAP students have also already taken BA, which is a spreadsheet-based introduction to business analytics (Isken 2003, 2014). The BA and PCDA courses have historically been offered in face-to-face mode in a computer teaching laboratory. Since 2020, these two courses have been also offered as online asynchronous courses. The newer AAP course has been offered since 2021 in an online asynchronous mode. All three courses have extensive course websites that are publicly accessible and include access to all of the video content and supporting files. These courses serve upper-level undergraduate and graduate students. Many are from the School of Business Administration, but they draw students from many programs across our campus.

What kind of Python background do students need to have in order to be ready for the EwP module? A basic familiarity with Python fundamentals for data analysis is required, including

- variables, arithmetic and Boolean operators, and basic data types;
- data structures, such as lists, dictionaries, tuples, NumPy arrays, and pandas data frames;
- flow control, such as branching with `if … else` and iterating with `for` and `while`;
- module imports;
- using built-in functions and accessing methods and properties of objects;
- creating functions;
- basic use of NumPy and pandas for data wrangling and analysis;
- basic plotting with matplotlib; and
- familiarity with Jupyter notebooks and an integrated development environment, such as Spyder, VSCode, or PyCharm.

The EwP module is made up of three submodules. The first of these focuses on building and using a typical spreadsheet model using Python. Then, the functionality developed in the first submodule is deployed as a reusable package. The final submodule is an introduction to using Python to manipulate Excel files.

All of the Python modules in the PCDA course as well as the entire AAP course are taught using a combination of Jupyter notebooks, Python scripts, and videos that walk readers through the notebooks. Jupyter notebooks, part of Project Jupyter, are an example of *literate computing* (Perez and Granger 2015) and allow the creation of rich interactive documents that support iterative exploration and development.

Central to Jupyter notebooks is a mix of *code cells*, *markdown cells*, and *output cells*. Code cells contain executable code, whereas markdown cells contain plain text using markdown for styling text. For example, Figure 1 shows two markdown cells separated by a code cell as they look when in editing mode. After

Figure 1. Notebook Cells in Edit Mode

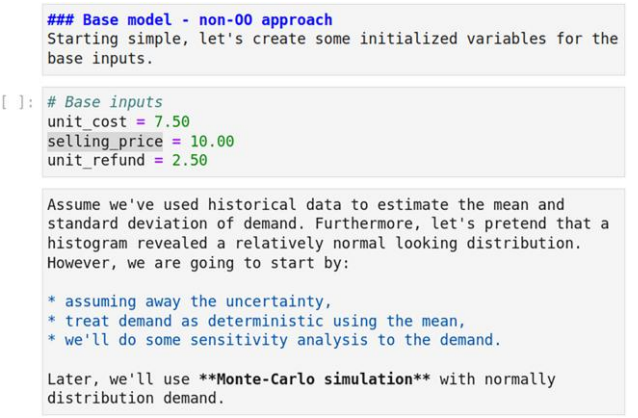
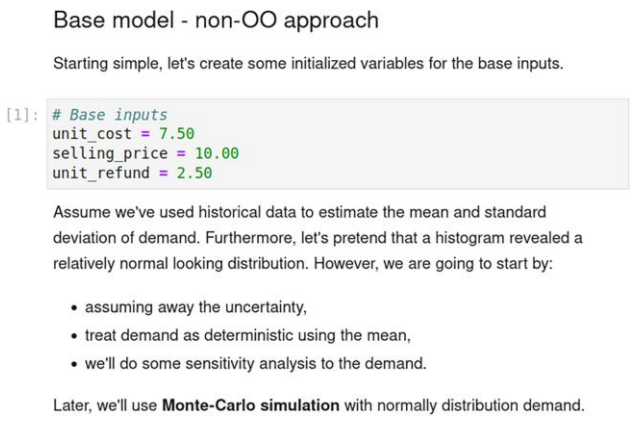


Figure 2. Notebook Cells After Execution



running the cells, they appear as shown in Figure 2. After executing a code cell, the output appears directly below it in an output cell. Mixing code and markdown cells makes it quite easy to create interactive tutorials, in which students not only can run prewritten code but also, can add their own code—all with the aid of supporting explanatory text. All of the blog posts upon which EwP is based were written in Jupyter notebooks.

The ability to weave standard explanatory language with formal programming languages has made notebook computing a popular tool in the data science education community (Perkel 2018). Jupyter notebooks can also include equations written in LATEX, data visualizations, and other multimedia. The technology is free and open source; runs on most computing platforms; and has a huge user community that has created a large number of resources for students, faculty, and industry practitioners. There are countless introductory tutorials for Jupyter notebooks, but the official Project Jupyter Documentation (Jupyter Team 2015) is a very good place to start if you are new to notebook computing. As with all tools, there are pros and cons to their use. See Rule et al. (2018), Barba et al. (2019), and Johnson (2020) for guidance on Jupyter notebook best practices for both research and teaching.

3. Submodule 1: What If Analysis with Python

There are two overarching learning objectives for this first EwP submodule. Students learn how Python might be used to do something they are quite familiar with doing in a spreadsheet—model building and sensitivity analysis. At the same time, they encounter more advanced Python programming concepts and techniques. In this way, the computational problem to be solved drives the introduction of more advanced Python concepts and techniques instead of such things being presented in a vacuum. See Figure 3 for how the objectives are presented to students via the course website.

I make it a point to show that there are often multiple possible approaches to a computational problem by being willing to abandon one approach and trying another. Emphasis is given to the lessons learned from each path explored. Every notebook is accompanied by screencasts that walk the students through the material and act as a guided tour with challenges to overcome along the way. Although all of the notebooks are publicly available (see Section 6), links to html versions of each of the notebooks are provided throughout the document, and readers are encouraged to use them while reading to see exactly how each topic is presented.

3.1. Notebook 1: Modeling and Data Tables (html version)

The stage is set with an overview of a typical spreadsheet-based model that most of the students have seen before. The notebook also includes a screenshot (Figure 4) of this Excel model.

> It's a really simple model in which we are selling a single product that we produce. There is a fixed cost to producing the product as well as a variable production cost per unit. We can sell the product for some price and we believe that demand for the product is related to the selling price through a power function. Let's assume for now that we have sufficient capacity to produce to demand and that all inputs are deterministic (we'll deal with simulating uncertainty later in this document).

Figure 3. Learning Objectives for Submodule 1

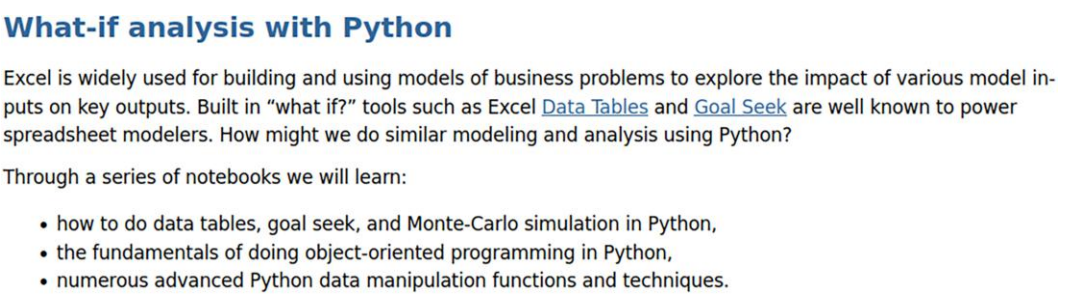
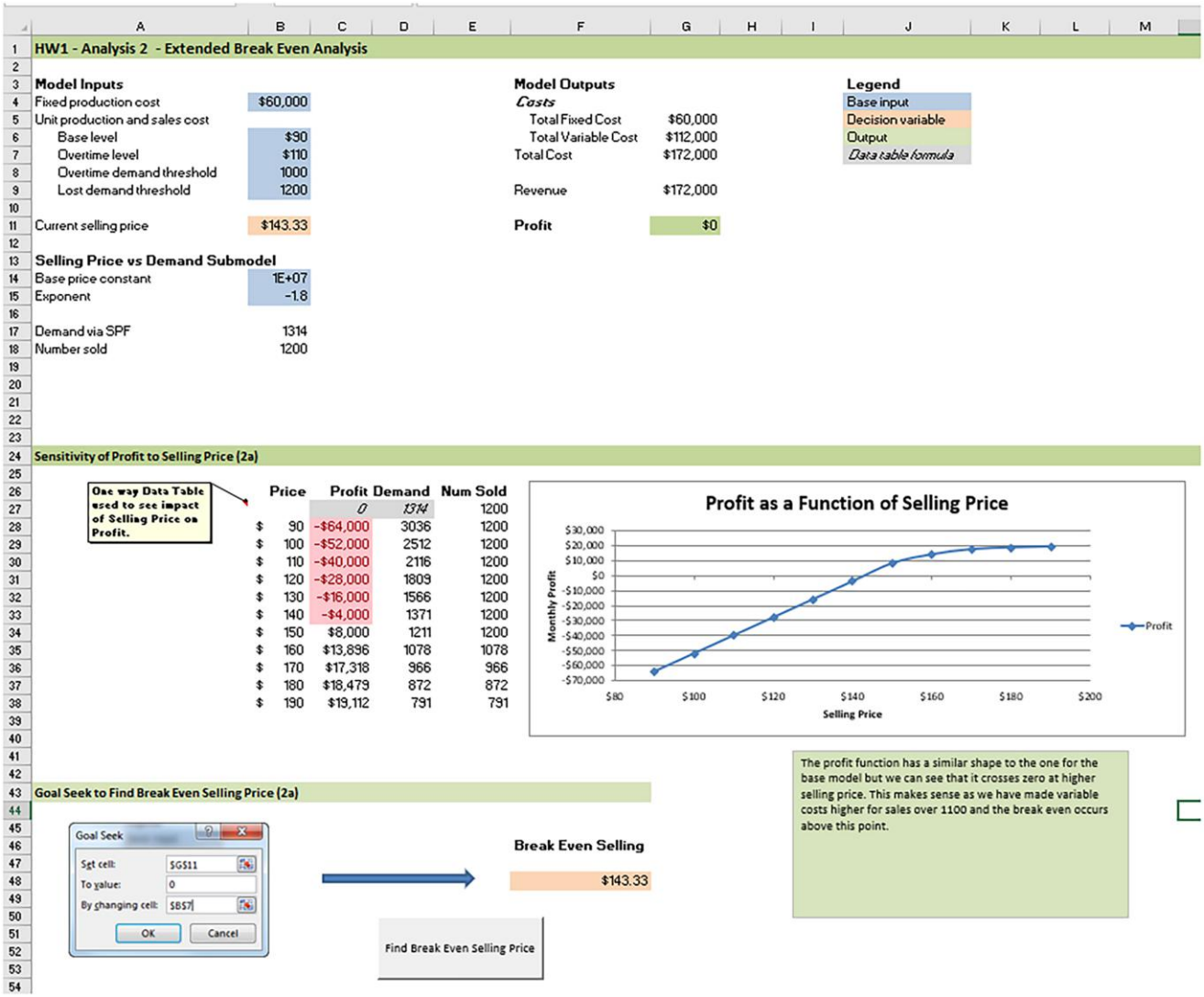


Figure 4. Typical Spreadsheet Model



The details aren't so important right now as is the overall structure of the model. There are a few key inputs and some pretty straightforward formulas for computing cost, revenue, and profit. Notice the 1-way Data Table being used to explore how profit varies for different selling prices. There's a graph driven by the Data Table and some Text Boxes used for annotation and summary interpretative comments. There's a button that launches Goal Seek to find the break even selling price and a 2-way Data Table (not shown) to explore the joint effect of selling price and variable cost. Classic Excel modeling stuff. How might we go about building a similar model using Python?

What if we wanted to push it a little further and model some of the key inputs with probability distributions to reflect our uncertainty about their values? In the Excel world, we might use add-ins such as @Risk which allow uncertain quantities to be directly modeled with probability distributions. For example, we might have a key input such as the exponent in the power function that relates selling price to demand that is highly uncertain. By modeling it with a probability distribution and then sampling from that distribution many times (essentially by recalcing the spreadsheet) we can generate a bunch of possible values for key outputs (e.g. profit) and use statistics to summarize these outputs using things like histograms and summary stats. Often this type of simulation model is referred to as a Monte-Carlo model to suggest repeated sampling from one or more probability distributions within an otherwise pretty static model. Again, how might we do this with Python?

We then introduce a similar modeling problem that will be the focus of the rest of the notebook. The Walton Bookstore problem appears in the chapter on Monte-Carlo simulation in textbooks that I have used in my

spreadsheet modeling course (Albright and Winston 2016, Winston and Albright 2018). The (slightly modified) problem is as follows.

- We have to place an order for a perishable product (e.g., a calendar).
- There is a known unit cost for each unit ordered.
- We have a known and fixed selling price.
- Demand is uncertain, but we can model it with some simple probability distribution.
- For each unsold item, we can get a partial refund of our unit cost.
- We need to select the order quantity for our one order for the year; orders can only be in multiples of 25.

The overall goal is to build a Python-based version of this model that will allow us to do sensitivity analysis, find the break-even point for demand, and analyze the problem using Monte-Carlo simulation.

3.1.1. Model Building Using a Procedural Programming Approach. We start with a very simple model that ignores the uncertainty in demand and does not use object-oriented programming concepts. Proceeding much like we would in Excel, base input values are stored in variables. Students are given skeleton code as scaffolding and must finish the expressions for computing key intermediate outputs (`order_cost`, `sales_revenue`, `refund_revenue`) and the final output—`profit`; see Figure 5.

Answers are provided at the bottom of the notebook so that students can attempt to finish the code but have a resource to correctly complete the code and move on. The screencast associated with the notebook also shows the code being completed in a subsequent code cell. Using skeleton code in this way has worked well both for laboratory-based versions of this type of course as well as for online asynchronous versions. For a laboratory-based course, students actively work on completing the code during class, and the instructor can move around the laboratory acting as a guide and consultant.

At this point, we discuss a fundamental difference between the computing model of Excel and that of

Figure 5. Instantiating Model Inputs and Creating Output Formulas

Starting simple, let's create some initialized variables for the base inputs.

[3]:
```
# Base inputs
unit_cost = 7.50
selling_price = 10.00
unit_refund = 2.50
```

Assume we've used historical data to estimate the mean and standard deviation of demand. Furthermore, let's pretend that a histogram revealed a relatively normal looking distribution. However, we are going to start by:

- assuming away the uncertainty,
- treat demand as deterministic using the mean,
- we'll do some sensitivity analysis to the demand.

Later, we'll use **Monte-Carlo simulation** with normally distribution demand.

[4]:
```
# Demand parameters
demand_mean = 193
demand_sd = 40

# Deterministic model
demand = demand_mean
```

Finally, let's set the initial order quantity. This will be the variable we'll focus on in the sensitivity analysis (along with demand). Of course, if we pretend that we know demand is really equal to 193, then we'd only consider ordering 175 or 200 and would pick the one leading to higher profit. Let's set it to 200.

[5]:
```
order_quantity = 200
```

Now we can compute the various cost and revenue components and get to the bottom line profit.

QUESTION 1: Complete the lines of code below (answer at bottom of notebook)

[ ]:
```
order_cost = unit_cost * order_quantity
sales_revenue = ??? * selling_price
refund_revenue = ??? * unit_refund
profit = sales_revenue + refund_revenue - order_cost
```

Jupyter notebooks. Although spreadsheets respond automatically to changes, Jupyter notebooks require rerunning all code cells that include or follow the changed code. This difference in behavior between Jupyter notebooks and spreadsheets might be one of the most important concepts for students to internalize. A well-known challenge, often referred to as the *hidden state problem*, associated with computational notebooks is the ease of running code cells out of order and introducing inadvertent errors; see Grus (2018) and Johnson (2020). This point (along with strategies to avoid it) is discussed during the first week of the course, which focuses on notebook computing best practices as well as a historical look at computational notebooks.

3.1.2. Sensitivity Analysis. Because order quantity is the key decision variable, we might want to see how profit changes for different order quantities. In Excel, the Data Table tool provides a familiar and easy way to do this. In a Data Table, a range of order quantities can be used as a row or column input, and one or more output values can be computed. In Python, the vectorized nature of the NumPy library (Harris et al. 2020) provides a simple way to accomplish the same thing. A range of order quantities is created as a NumPy array that can be used directly to compute vectors of all of the intermediate and final output variables. Much like Data Tables, vectorized computations avoid having to explicitly iterate, or loop, through a collection of input values.

A natural next step is to do the equivalent of an Excel two-way Data Table. By creating a profit function that takes all of the base inputs as arguments, a list comprehension can then be used to generate the equivalent of a two-way Data Table. Even better, this actually allows us to do the equivalent of an *n*-way Data Table with an arbitrary number of outputs—something Excel does not have. This example provides an effective way to show the power of creating reusable functions and Python's handy list comprehension construct. As shown in Figure 6, a few code and markdown cells are used to create the code, provide some explanation, and show the output. This is the same output one would get with an Excel two-way

Figure 6. A Two-Way Data Table

[14]:
```
def bookstore_profit(unit_cost, selling_price, unit_refund, order_quantity, demand):
    '''
    Compute profit in bookstore model
    '''
    order_cost = unit_cost * order_quantity
    sales_revenue = np.minimum(order_quantity, demand) * selling_price
    refund_revenue = np.maximum(0, order_quantity - demand)
    profit = sales_revenue + refund_revenue - order_cost
    return profit
```

A list comprehension can be used to create an n-way Data Table. List comprehensions provide a nice syntactical shortcut to creating a list based one looping over one or more iterables and computing associated values to store in the list. As a simple first example that's related to the model we are working on, let's compute the difference between demand and order quantity for all the combinations of those variables in the following ranges:

[15]:
```
demand_range = np.arange(50, 301, 5)
order_quantity_range = np.arange(50, 301, 25)
```

[16]:
```
# Create data table (as a list of tuples)
data_table_1 = [(d, oq, bookstore_profit(unit_cost, selling_price, unit_refund, oq, d))
                for d in demand_range for oq in order_quantity_range]

# Convert to dataframe
dtbl_1_df = pd.DataFrame(data_table_1, columns=['Demand', 'OrderQuantity', 'Profit'])
print(dtbl_1_df.head(5))
```

```
   Demand  OrderQuantity  Profit
0      50             50   125.0
1      50             75   -37.5
2      50            100  -200.0
3      50            125  -362.5
4      50            150  -525.0
```

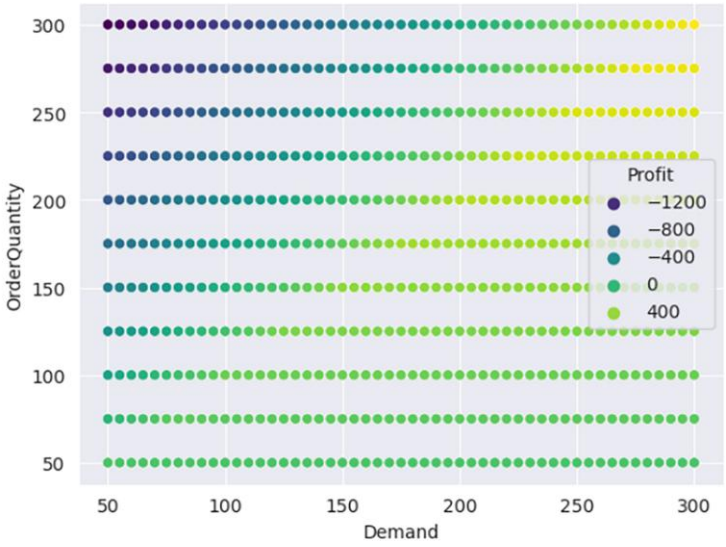
[17]:
```
print(dtbl_1_df.tail(5))
```

```
     Demand  OrderQuantity  Profit
556     300            200   500.0
557     300            225   562.5
558     300            250   625.0
559     300            275   687.5
560     300            300   750.0
```

Figure 7. Scatterplot Based on Two-Way Data Table

Here's a quick scatter plot showing how profit (mapped to color hue) varies jointly by order quantity and demand.
No surprises.

```
[18]: sns.set_style("darkgrid")
      sns.scatterplot(x="Demand", y="OrderQuantity", hue="Profit",
                      data=dtbl_1_df, palette="viridis");
```



Data Table, just laid out differently. Then, in Figure 7, we show how the output can be turned into a scatterplot with profit mapped to a color hue.

3.1.3. The Object Nature of Python. It is not clear whether this nonobject-oriented procedural approach will end up making the most sense for doing spreadsheet-type modeling in Python. This uncertainty is made explicit in the notebook, and students are led on a bit of a journey of discovery as we explore different software designs for this modeling problem. I purposefully designed the notebooks in this way to show students that the path to a solution is not necessarily linear and that giving yourself the freedom to explore alternatives, many of which will be dead ends, can lead to a rich learning experience. It is also more realistic in the sense that this is how most real problems are solved. It is analogous to the idea that mathematical proofs usually do not reflect the myriad of failed approaches that were tried but that ultimately contributed to the final product.

In preparation for building an object-oriented version of the model, basic object concepts are reviewed. Because most students have done some Excel VBA programming in a previous course, I refer back to a few properties and methods of the Excel `Worksheet` object. In Python, everything is an object. Using Python lists as an example, we explore some of its attributes, such as methods for appending items to a list and reversing the order of its elements. The `dir` function is used to see all of an object's attributes, and we see that even things like integers are objects in Python. With this basic object refresher, we are ready to do some actual object-oriented programming and create our own classes.

3.1.4. Creating an Object-Oriented Model. For our initial design of a BookstoreModel class, we make all of the base inputs class attributes (properties) and then add method attributes to compute outputs, such as costs, revenues, and profits; see Figure 8. Students can easily see the mapping between elements in the initial procedural model and this new object-oriented version. Several important concepts and syntactical details of doing object-oriented programming Python are highlighted. Again, the goal is to weave in these more advanced Python programming concepts within the context of a very familiar modeling problem. This section of the notebook ends with using the `BookstoreModel` class to create a new model object instance with all of its base inputs instantiated with values. Before moving on, students are presented with a challenge involving enhancements to the `BookstoreModel` class. Interspersing short coding challenges throughout the notebook gives students a chance to test themselves on their understanding and to get some coding practice.

A few design dilemmas are posed and some nonworking approaches are illustrated as we try to decide on how to implement an n-way data table function for

Figure 8. Object-Oriented Version of the Model

```
[46]: class BookstoreModel():
          def __init__(self, unit_cost, selling_price, unit_refund, order_quantity, demand):
              self.unit_cost = unit_cost
              self.selling_price = selling_price
              self.unit_refund = unit_refund
              self.order_quantity = order_quantity
              self.demand = demand

          def order_cost(self):
              """Compute total order cost"""
              return self.unit_cost * self.order_quantity

          def sales_revenue(self):
              """Compute sales revenue"""
              return np.minimum(self.order_quantity, self.demand) * self.selling_price

          def refund_revenue(self):
              """Compute revenue from refunds for unsold items"""
              return np.maximum(0, self.order_quantity - self.demand) * self.unit_refund

          def profit(self):
              '''
              Compute profit
              '''
              profit = self.sales_revenue() + self.refund_revenue() - self.order_cost()
              return profit
```

this new object-oriented model. These metareflections are intentionally included to better mimic actual software development and reinforce the idea that software development is much more than simply writing code to implement a perfectly thought-out design. We forge ahead with some preliminary ideas and are confronted with an intermediate problem related to generating a list of dictionaries that represent combinations of inputs, or *scenarios*, to evaluate. This provides a great opportunity to remind students of one of the great strengths of the open-source software ecosystem; we can leverage work done in other packages and even look at actual source code to see how something was implemented. In this case, the well-known scikit-learn package (Pedregosa et al. 2011) has a `ParameterGrid` function that solves our scenario generation problem quite nicely. This example also reminds the students of the value of being able to make sense of API documentation and to understand object-oriented code written by others.

This first notebook draws to a close with the creation of a `data_table` function (Figure 9) that allows us to do sensitivity analysis with the objected-oriented model. The payoff is the creation of the faceted plot in Figure 10 showing how profit varies for different order quantities and demand levels. We recap the main things learned in this notebook and prepare to add goal-seeking capability to our model.

3.2. Notebook 2: Goal Seek (html version)

No self-respecting modeling tool would be complete without goal-seeking capability to do things such as finding the break-even demand point in the bookstore model. For many students, Excel's Goal Seek tool can seem almost magical when encountered for the first time. Even after they get a sense of what it is doing, understanding the importance of the initial solution guess and how it can lead to Goal Seek reporting different solutions is not very transparent. With Python, students can gain a much better understanding of how tools, like Goal Seek, really work.

Before trying to build our `goal_seek` function, we take a brief detour into basic root-finding algorithms through packages like SciPy (Virtanen et al. 2020) as well as in open-source notebooks and scripts that are widely available in the Python data science ecosystem (Walls 2023). We compare the output of Excel's Goal Seek with different starting values with the values obtained by different root-finding algorithms implemented in Python. The importance of starting values and also, the range of different root-finding algorithms available are often surprising to business students who have spent most of their analytical life in Excel.

A `goal_seek` function is created that implements a simple bisection search, and we use it to find the break-even demand point in our bookstore model. See Figure 11. With `data_table` and `goal_seek` functions implemented, we are ready to move on to doing Monte-Carlo simulation.

3.3. Notebook 3: Monte-Carlo Simulation (html version)

In the Excel modeling world, Monte-Carlo simulation can be done without add-ins, but packages, like @Risk,

Figure 9. An *n*-Way Data Table Function

```
[74]: def data_table(model, scenario_inputs, outputs):
          '''Create n-inputs by m-outputs data table.

          Parameters
          ----------
          model : object
              User defined object containing the appropriate methods and properties for computing outputs from inputs
          scenario_inputs : dict of str to sequence
              Keys are input variable names and values are sequence of values for each scenario for this variable.
          outputs : list of str
              List of output variable names

          Returns
          -------
          results_df : pandas DataFrame
              Contains values of all outputs for every combination of scenario inputs
          '''

          # Clone the model using deepcopy
          model_clone = copy.deepcopy(model)

          # Create parameter grid
          dt_param_grid = list(ParameterGrid(scenario_inputs))

          # Create the table as a list of dictionaries
          results = []

          # Loop over the scenarios
          for params in dt_param_grid:
              # Update the model clone with scenario specific values
              model_clone.update(params)
              # Create a result dictionary based on a copy of the scenario inputs
              result = copy.copy(params)
              # Loop over the list of requested outputs
              for output in outputs:
                  # Compute the output.
                  out_val = getattr(model_clone, output)()
                  # Add the output to the result dictionary
                  result[output] = out_val

              # Append the result dictionary to the results list
              results.append(result)

          # Convert the results list (of dictionaries) to a pandas DataFrame and return it
          results_df = pd.DataFrame(results)
          return results_df
```


can make the process much easier. In particular, random variable generation for a wide range of probability distributions is facilitated by add-ins, like @Risk. In Python, the NumPy package provides random variate generation for a large number of distributions, and similarly, SciPy provides functions for computing distribution-related quantities for many probability distributions. Much like we did when creating the `data_table` function, we start with adding uncertainty to a single input variable in the bookstore model and rely on NumPy's vectorized computing capabilities to generate the profit associated with each realization of the random input. Then, we can do standard statistical analysis of the simulation output using Python packages, such as pandas and SciPy. In spreadsheet-based simulation textbooks, the notion of the "Flaw of Averages" (Savage 2012) is often illustrated. We do the same with our Python-based model, showing how replacing random demand by its mean can result in a wildly optimistic estimate of the mean profit found through analyzing the simulation output. Much like doing simulation in Excel without add-ins, using Python makes transparent the quite simple and brute force nature of Monte-Carlo simulation.

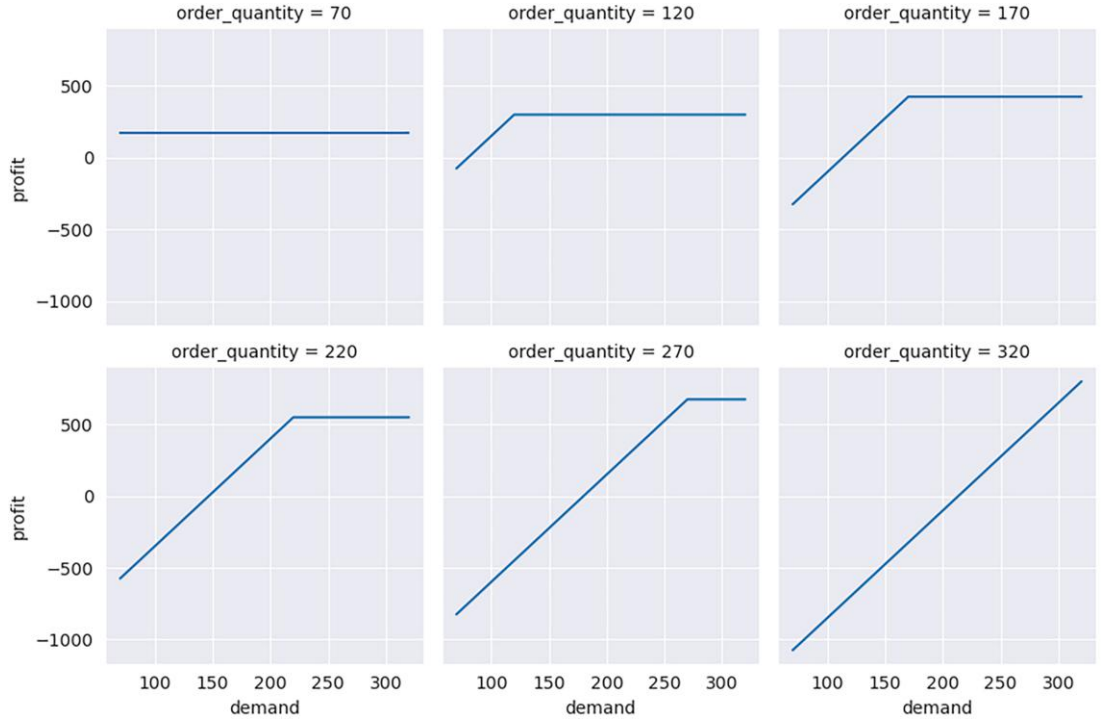
Moving on to multiple uncertain inputs raises challenges similar to those we faced when moving from a one-way to an *n*-way data table function. In addition to modeling multiple uncertain inputs, we want to be able to specify a range of scenarios to run, just as we did for the `data_table` function. Figure 12 shows the design specifications for our `simulate` function.

This is followed by a well-commented first attempt at a `simulate` function. By this point, students are well equipped to understand the code as it builds on basic Python concepts and more advanced concepts that we have already covered in the EwP module. See Figure 13.

The rest of the notebook is spent trying out the `simulate()` function and ends with creating a grouped box plot (see Figure 14) along with a faceted plot of profit histograms created with the popular Seaborn package.

The first submodule and its three notebooks are very much about the type of modeling typically done in a spreadsheet-based course. In a spreadsheet-based course, students learn numerous Excel functions and techniques in the process of building and exercising such models. Similarly, numerous Python techniques are scattered throughout the three modeling-focused

Figure 10. Faceted Plot Based on a Two-Way Data Table



notebooks. Now, we shift gears and move beyond what might be considered prototyping and into the realm of deployment. Figure 15 shows how the notebook ends and gives students a preview of the next submodule.

4. Submodule 2: The Packaging and Documentation Notebooks

Now that we have created some useful modeling functionality, our goal is to make it easier for a modeler to actually use the code. Certainly, we do not want to rely on copying, pasting, and modifying code when building a new model. In the Python world, the natural next step is to deploy our code as a Python *package* (Beuzen and Timbers 2022) that can then be imported like any other Python library. Figure 16 shows the learning objectives from this submodule page on the course website.

Creating a deployable Python package is a good exercise in designing and creating a reusable software artifact. Thinking about how others will interact with your software usually leads to better-designed software. In this submodule, we create a Python package that can be imported and used by other modelers. This will require some design changes to our software and will also involve moving code out of Jupyter notebooks and into Python script files. Although Jupyter notebooks can be integrated into production environments (Ufford et al. 2019), it is much more common for code to live in Python script files. In addition to creating a deployable package, this submodule also discusses the importance of documentation in its many forms in a Python project.

4.1. Notebook 4: Project Packaging (html version)

The first notebook in this submodule starts by explaining what Python packages are and their role in sharing code. We then revisit a concept from earlier in the course—creating a good project folder structure. For this, we use what is known as a cookie cutter that can automatically generate a project folder structure and key files from the answers to a few prompts (Roy and Cookiecutter Community 2012). There are a few different folder layouts used in Python projects, and there is a bit of discussion on the folder structure we will use and resources given for exploring this issue further. Students are reminded of the importance of version control, and we initialize a git repository for our budding package. The basics of version control were already covered in the first week of the course. My experience is that most students have little exposure to software project management concepts, such as project folder structures and version control. This can lead to a jumble of files with incoherent names used as a proxy for version control—`whatif_v1.py`, `whatif_v2.py`, `whatif_final.py`, `whatif_finalfinal.py`, and so on.

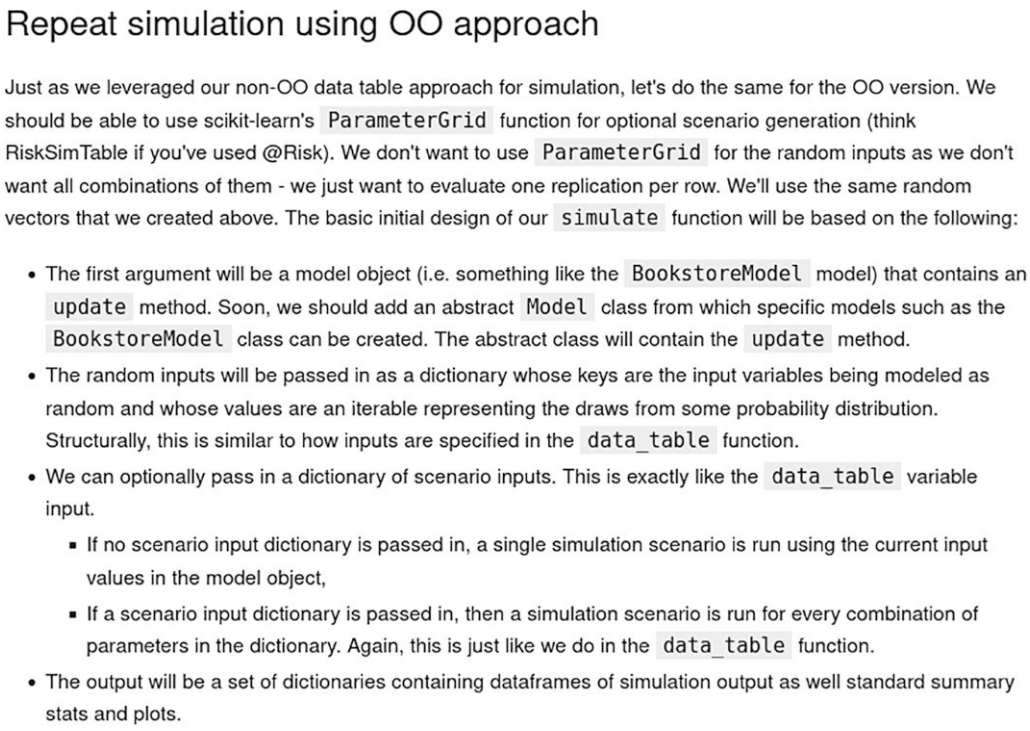
Key software design changes are then made, which will make the code easier to use. These changes include

Figure 11. The goal_seek Function

```
def goal_seek(model, obj_fn, target, by_changing, a, b, N=100, verbose=False):
    '''Approximate solution of f(x)=0 on interval [a,b] by bisection method.

    Parameters
    ----------
    model : object
        User defined object containing the appropriate methods and properties for doing the desired goal seek
    obj_fn : function
        The function for which we are trying to approximate a solution f(x)=target.
    target : float
        The goal
    by_changing : string
        Name of the input variable in model
    a,b : numbers
        The interval in which to search for a solution. The function returns
        None if (f(a) - target) * (f(b) - target) >= 0 since a solution is not guaranteed.
    N : (positive) integer
        The number of iterations to implement.
    verbose : boolean (default=False)
        If True, root finding progress is reported

    Returns
    -------
    x_N : number
        The midpoint of the Nth interval computed by the bisection method.
        If all signs of values f(a_n), f(b_n) and f(m_n) are the same at any
        iteration, the bisection method fails and return None.
    '''
    # Clone the model
    model_clone = copy.deepcopy(model)

    # The following bisection search is a direct adaptation of
    # https://www.math.ubc.ca/~pwalls/math-python/roots-optimization/bisection/
    setattr(model_clone, by_changing, a)
    f_a_0 = getattr(model_clone, obj_fn)()
    setattr(model_clone, by_changing, b)
    f_b_0 = getattr(model_clone, obj_fn)()

    if (f_a_0 - target) * (f_b_0 - target) >= 0:
        # print("Bisection method fails.")
        return None

    # Initialize the end points
    a_n = a
    b_n = b
    for n in range(1, N+1):
        # Compute the midpoint
        m_n = (a_n + b_n)/2

        # Function value at midpoint
        setattr(model_clone, by_changing, m_n)
        f_m_n = getattr(model_clone, obj_fn)()

        # Function value at a_n
        setattr(model_clone, by_changing, a_n)
        f_a_n = getattr(model_clone, obj_fn)()

        # Function value at b_n
        setattr(model_clone, by_changing, b_n)
        f_b_n = getattr(model_clone, obj_fn)()

        if verbose:
            print(f"n = {n}, a_n = {a_n}, b_n = {b_n}, m_n = {m_n}, width = {b_n - a_n}")

        # Figure out which half the root is in, or if we hit it exactly, or if the search failed
        if (f_a_n - target) * (f_m_n - target) < 0:
            a_n = a_n
            b_n = m_n
        elif (f_b_n - target) * (f_m_n - target) < 0:
            a_n = m_n
            b_n = b_n
        elif f_m_n == target:
            return m_n
        else:
            return None

    # If we get here we hit iteration limit, return best solution found so far
    return (a_n + b_n)/2
```

creating an abstract `Model` base class from which different model classes can be created and moving the `data_table`, `goal_seek`, and `simulate` functions into the `Model` base class as methods. All of the code is moved into a code module named `whatif.py`. Students are now ready to learn about the basics of turning our code into an installable package. Options are presented for installing our package locally so that we can use it for different modeling projects. This includes an explanation of how and where Python searches for packages when import statements are encountered in code. There is much complexity lurking here, and students are reminded that they will need to revisit this topic frequently. I have made every effort to distill

Figure 12. Design Specification for `simulate()` Function



Repeat simulation using OO approach

Just as we leveraged our non-OO data table approach for simulation, let's do the same for the OO version. We should be able to use scikit-learn's `ParameterGrid` function for optional scenario generation (think RiskSimTable if you've used @Risk). We don't want to use `ParameterGrid` for the random inputs as we don't want all combinations of them - we just want to evaluate one replication per row. We'll use the same random vectors that we created above. The basic initial design of our `simulate` function will be based on the following:

- The first argument will be a model object (i.e. something like the `BookstoreModel` model) that contains an `update` method. Soon, we should add an abstract `Model` class from which specific models such as the `BookstoreModel` class can be created. The abstract class will contain the `update` method.
- The random inputs will be passed in as a dictionary whose keys are the input variables being modeled as random and whose values are an iterable representing the draws from some probability distribution. Structurally, this is similar to how inputs are specified in the `data_table` function.
- We can optionally pass in a dictionary of scenario inputs. This is exactly like the `data_table` variable input.
  - If no scenario input dictionary is passed in, a single simulation scenario is run using the current input values in the model object,
  - If a scenario input dictionary is passed in, then a simulation scenario is run for every combination of parameters in the dictionary. Again, this is just like we do in the `data_table` function.
- The output will be a set of dictionaries containing dataframes of simulation output as well standard summary stats and plots.

things down to a minimal level of complexity needed to convey the important points.

This notebook ends by pointing the students to another notebook in which our newly deployed whatif package is used for a completely different model—the New Car Simulation that has been a part of Winston and Albright's textbooks for many years (Albright and Winston 2016, Winston and Albright 2018). This new model differs structurally from the BookstoreModel but is representative of many financial models in that cash flows over multiple years need to be computed and summarized. This helps cement the idea that there is value in creating reusable code.

4.2. Notebook 5: Documentation (html version)

The deployment-focused submodule ends with a short notebook discussing the different types of documentation needed in a typical project. This includes code comments, docstrings, `readme` files, and generating documentation by creating reStructuredText files (similar to but more powerful than markdown) and using Sphinx, the Python documentation generation tool. This is one of those tasks that few people like to do but can be very important for the long-term success of a project. Even if you are just creating a tool for yourself, you will forget the details over time, and documentation can be very beneficial in refreshing your memory.

5. Submodule 3: Excel Data Wrangling with Python

The EwP module ends with what has become a relatively common use of Python with Excel—automating various Excel data-wrangling tasks. Examples include the following.

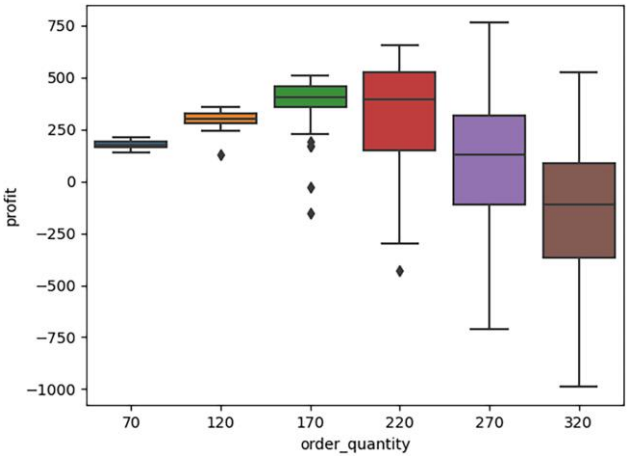
- You have a whole folder full of csv (or Excel) files with the same file structure, and you need to combine them into a single file. You might also need to make some changes to the consolidated file.
- You have an Excel file with multiple sheets of similarly structured data, and you want to consolidate them into a single sheet.
- You have an Excel file with data in wide format, and you need to convert it to long format and then, perhaps export out individual files (one per the key column(s) in the long formatted data).
- You have an Excel file acting as a simple flat-file database. Periodically, you get new Excel files that need to get appended to the "database" file.

A single Jupyter notebook is used to give a taste of using Python to automate the process of working with Excel files by tackling each of the four examples. Each of these examples is based on a real problem I encountered either in research or in industrial projects. Several web-based resources are shared, including the Practical Business Python blog written by Chris Moffitt, a data analytics professional (Moffitt 2022). This blog, which

Figure 13. The simulate Function

```
def simulate(model, random_inputs, outputs, scenario_inputs=None, keep_random_inputs=False):
    '''Simulate model for one or more scenarios

    Parameters
    ----------
    model : object
        User defined object containing the appropriate methods and properties for computing outputs from inputs
    random_intputs : dict of str to sequence of random variates
        Keys are stochastic input variable names and values are sequence of £nf random variates, where £nf is the number of simulation replications
    outputs : list of str
        List of output variable names
    scenario_inputs : optional (default is None), dict of str to sequence
        Keys are deterministic input variable names and values are sequence of values for each scenario for this variable. Is consumed by
        scikit-learn ParameterGrid() function. See https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.ParameterGrid.html
    keep_random_inputs : optional (default is False), boolean
        If True, all the random input variates are included in the results dataframe

    Returns
    -------
    results_df : list of dict
        Values of all outputs for each simulation replication. If `scenario_inputs` is not None, then this is also for every combination of scenario inputs
    '''

    # Clone the model
    model_clone = copy.deepcopy(model)

    # Update clone with random_inputs
    model_clone.update(random_inputs)

    # Store raw simulation input values if desired
    if keep_random_inputs:
        scenario_base_vals = vars(model_clone)
    else:
        scenario_base_vals = vars(model)

    # Initialize output counters and containers
    scenario_num = 0
    scenario_results = []

    # Check if multiple scenarios
    if scenario_inputs is not None:
        # Create parameter grid for scenario inputs
        sim_param_grid = list(ParameterGrid(scenario_inputs))

        # Scenario loop
        for params in sim_param_grid:
            model_clone.update(params)
            # Initialize scenario related outputs
            result = {}
            scenario_vals = copy.copy(params)
            result['scenario_base_vals'] = scenario_base_vals
            result['scenario_num'] = scenario_num
            result['scenario_vals'] = scenario_vals
            raw_output = {}

            # Output measure loop
            for output_name in outputs:
                output_array = getattr(model_clone, output_name)()
                raw_output[output_name] = output_array

            # Gather results for this scenario
            result['output'] = raw_output
            scenario_results.append(result)
            scenario_num += 1

        return scenario_results

    else:
        # Similar logic to above, but only a single scenario
        results = []
        result = {}

        result['scenario_base_vals'] = scenario_base_vals
        result['scenario_num'] = scenario_num
        result['scenario_vals'] = {}

        raw_output = {}
        for output_name in outputs:
            output_array = getattr(model_clone, output_name)()
            raw_output[output_name] = output_array

        result['output'] = raw_output
        results.append(result)

        return results
```

Figure 14. Profit Box Plots by Order Quantity



focuses on helping Excel-centric analytics professionals transition some of their work to Python, includes numerous posts on different aspects of Python and Excel integration.

There are a number of Python-based tools that either include functionality for working with Excel files or are dedicated to specific Excel-related operations. These include

- pandas—read_excel, ExcelWriter, to_excel—reading and writing Excel files;
- openpyxl—can read, write, and modify Excel files;
- XlsxWriter—create new Excel files but cannot edit existing Excel files; and
- xlwings—manipulate the Excel application from Python.

5.1. Example 1: Concatenating Many csv Files

Although this first example is not necessarily Excel related, using Excel to consolidate csv files through a series of manual copy-paste operations is something that many Excel users will admit to having done. After getting familiar with `pathlib`, a library for working with file systems that has recently been added to Python's standard library, students are led through the development of a short procedure that creates a list of the csv files to concatenate, uses pandas to read and combine the files in a `DataFrame`, and writes out the consolidated data as a csv file.

5.2. Example 2: Consolidating Data from Multiple Sheets in a Excel File Using Pandas

Two different approaches are shown to consolidate data from multiple sheets into a single sheet. One approach uses pandas, whereas the other relies on a library specifically created for working with Excel files from Python, openpyxl. One advantage to using Python rather than Visual Basic for Applications for this task is that Python is well suited for working with data in many different formats, including csv, JSON, XML, and a myriad of database formats. The Excel file may be just one data source out of a collection of data sources that are used to create some sort of combined database. The generality of Python is a strength in such scenarios.

5.3. Example 3: Dealing with Wide Data and a Multirow Header

Reshaping, or tidying, data (Wickham 2014) from wide to long format is a common data-wrangling task, especially in preparation for plotting. In this example, not only do students have to grapple with reshaping the data in an Excel workbook, but they also have to deal with the all too common problem of multirow header lines in a spreadsheet. Excel's flexibility makes it easy to create such multirow headers and pity the poor analyst who then has to deal with it when trying to move the data into some sort of data frame or database table. This example is posed as a challenge to the students. Examples of the desired outputs are shown, and a few hints are given. The answer is provided at the end of the notebook. This example really shows the power of Python in creating a reproducible and automated approach to dealing with a poorly structured Excel workbook, creating not only restructured data frames but also nontrivial plots. Attempting to do all of this in Excel, even with VBA, is not an easy task.

Figure 15. Closing Thoughts in the Simulation Notebook

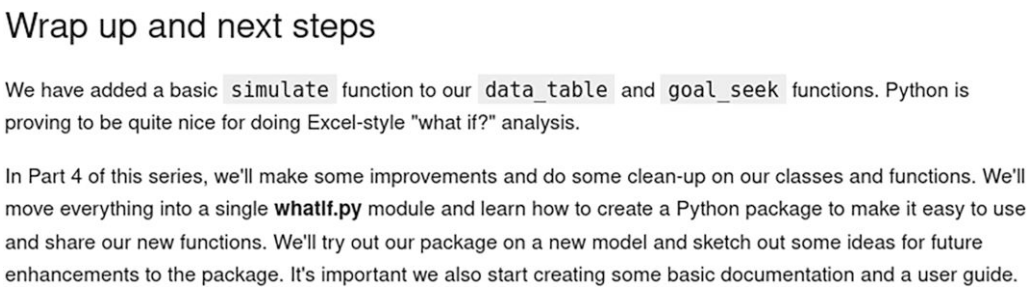
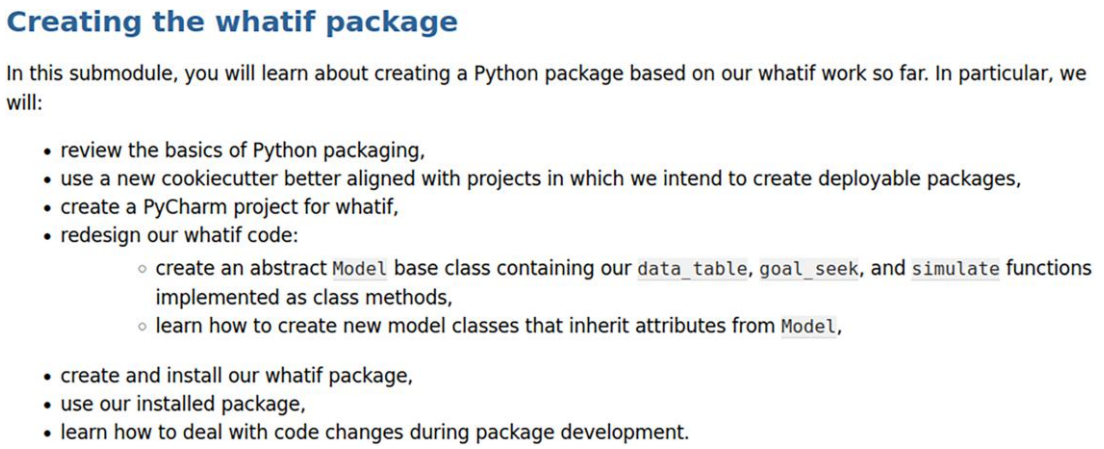


Figure 16. Learning Objectives for Submodule 2



5.4. Example 4: Appending New Spreadsheet Data in a Consolidated Excel Workbook

Excel is not a database. Of course, that does not stop people from using it as a simple flat-file database. In such a scenario, it is not uncommon to have to periodically append new data to the end of a range of data in an Excel workbook. Again, a combination of openpyxl and pandas allows us to meet this challenge.

5.5. More on Python/Excel Integration

The examples barely scratch the surface of the use cases for Python integration with Excel. Our notebook ends with a shout out to the highly regarded Practical Business with Python blog and a list of specific examples that students can explore on their own:

- combining data from multiple Excel files—file globbing, concatenating data frames;
- reading poorly structured Excel files with pandas—advanced use of `read_excel`, accessing ranges and tables;
- common Excel tasks demonstrated in pandas—totals rows, fuzzy string matching;
- common Excel tasks demonstrated in pandas part 2—selection and filtering;
- improving pandas Excel output—using XlsxWriter to format Excel workbooks from Python;
- creating advanced Excel workbooks—XlsxWriter, inserting VBA from Python(!), using COM to merge sheets; and
- interactive data analysis with Python and Excel—using xlwings to "glue" Python and Excel together, using sqlalchemy to interact with databases.

6. Software Availability and Classroom Use

Each of the submodules described has its own dedicated course web page. Links to each submodule page can be found on this EwP landing page within the AAP course website. Each page contains learning objectives, a link to a compressed file containing the Jupyter notebooks and other supporting files, a set of activities with accompanying screen casts, and links to additional web-based materials to explore. The source reStructuredText files for the course web pages can be found at this AAP GitHub repository. There is also a whatif GitHub repository for the `whatif` package. It contains the original notebooks used in the blog posts that motivated this project.

This module has been included in my Advanced Analytics with Python course for the past three years. The course is offered in a seven-week accelerated format during the summer in an online asynchronous mode. The EwP module is covered over the course of two weeks and is the subject of one of the major homework assignments for the semester. The first part of the assignment involves creating a basic model, similar in complexity to the BookstoreModel, and doing sensitivity analysis, goal seeking, and simulation with the model. It requires the students to import and use the `whatif.py` package we created as part of the module. The second part of the assignment is based on the last submodule and requires students to complete a task involving the creation of a multisheet Excel workbook from a set of csv files using Python. Then, using the openpyxl library, they have to create and add formulas to the workbook and format them properly. The ranges used in the formulas vary by sheet, and they require students to determine of the number of rows in each sheet of data and use that information to construct the appropriate formula. Students must create a proper project folder structure and put their project under version control. An example assignment is available from here. The module has been well received, and several of the working students have reported back to me that they have found use cases for these techniques in their day-to-day analytical activities.

7. The Future of Python and Excel

Rumors have swirled for a number of years that Microsoft was considering adding Python as a first-class

language alternative to VBA within their MS Office suite of packages (Cimpanu 2017). During the review process for this article, this development finally materialized, albeit in a limited roll out to beta testers. This has potentially enormous implications for the future of both Excel and Python. At this point, it is probably safe to say that this development further underscores the importance of business analytics students adding Python to their analytical toolbox.

There continues to be significant interest in using Python to manipulate not only Excel workbooks but other Office documents, such as PowerPoint presentations (Canny 2013). Given that the ubiquity of both Excel and PowerPoint in business education and practice, inclusion of this material in a business school Python-based analytics course seems quite appropriate. Students can learn more advanced and extremely useful Python concepts and techniques within a context with which they are intimately familiar. Manipulation of MS Office documents is directly relevant to the job content of many of our students. This teaching module is a natural complement to the commonly taught spreadsheet-based analytics and introductory data science courses found in many business schools and provides an interesting way to teach and learn Python for analytics.

References

Albright SC, Winston WL (2016) *Business Analytics: Data Analysis & Decision Making*, 6th ed. (Cengage Learning, Boston).

Alderson DL (2022) Interactive computing for accelerated learning in computation and data science. *INFORMS Trans. Ed.* 22(2): 130–145.

Babier A, Fernandes C, Zhu IY (2023) Advising student-driven analytics projects: A summary of experiences and lessons learned. *INFORMS Trans. Ed.* 23(2):121–135.

Baker K (2000) Gaining insight in linear programming from patterns in optimal solutions. *INFORMS Trans. Ed.* 1(1):4–17.

Barba LA, Barker LJ, Blank DS, Brown J, Downey AB, George T, Heagy LJ, et al. (2019) Teaching and learning with Jupyter. Accessed July 28, 2023, https://jupyter4edu.github.io/jupyter-edu-book.

Bell PC (2000) Teaching business statistics with Microsoft Excel. *INFORMS Trans. Ed.* 1(1):18–26.

Beuzen T, Timbers T (2022) Welcome to Python packages!—Python packages. Accessed July 28, 2023, https://py-pkgs.org/.

Camm JD, Cochran JJ, Fry MJ, Ohlmann JW (2020) *Business Analytics* (Cengage Learning, Boston).

Canny S (2013) Python-pptx. Accessed July 28, 2023, https://python-pptx.readthedocs.io/en/latest/.

Carraway RL, Clyman DR (2000) Integrating spreadsheets into a case-based MBA quantitative methods course: Real managers make real decisions. *INFORMS Trans. Ed.* 1(1):38–46.

Cimpanu C (2017) Microsoft considers adding Python as an official scripting language to Excel. Accessed July 28, 2023, https://www.bleepingcomputer.com/news/microsoft/microsoft-considers-adding-python-as-an-official-scripting-language-to-excel/.

Evans JR (2000) Spreadsheets as a tool for teaching simulation. *INFORMS Trans. Ed.* 1(1):27–37.

Grus J (2018) I don't like notebooks. *JupyterCon 2018.*

Harris CR, Millman KJ, van der Walt SJ, Gommers R, Virtanen P, Cournapeau D, Wieser E, et al. (2020) Array programming with NumPy. *Nature* 585(7825):357–362.

Isken MW (2003) Data cleansing and analysis as a prelude to model based decision support. *INFORMS Trans. Ed.* 3(3):23–75.

Isken MW (2014) Translating a laboratory based spreadsheet modeling course to an online format: Experience from a natural experiment. *INFORMS Trans. Ed.* 14(3):120–128.

Johnson JW (2020) Benefits and pitfalls of Jupyter notebooks in the classroom. *Proc. 21st Annual Conf. Inform. Tech. Ed.*, Association for Computing Machinery, New York, 32–37.

Jupyter Team (2015) Project Jupyter documentation. Accessed July 28, 2023, https://docs.jupyter.org/en/latest/.

Lakhani S, Vora M, Mahajan A (2023) Puzzle—An OR approach for wordle. *INFORMS Trans. Ed.* 24(1):103–104.

McKinney W (2022) *Python for Data Analysis* (O'Reilly, Sebastopol, CA).

Moffitt C (2022) Practical business Python. Accessed July 28, 2023, https://pbpython.com/.

Nelson C (2024) *Software Engineering for Data Scientists* (O'Reilly, Sebastopol, CA).

Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, Blondel M, et al. (2011) Scikit-learn: Machine learning in Python. *J. Machine Learn. Res.* 12:2825–2830.

Perez F, Granger BE (2015) Project Jupyter: Computational narratives as the engine of collaborative data science. *Jupyter Blog* (July 7), https://blog.jupyter.org/project-jupyter-computational-narratives-as-the-engine-of-collaborative-data-science-2b5fb94c3c58.

Perkel JM (2018) Why Jupyter is data scientists' computational notebook of choice. *Nature* 563(7732):145–147.

Powell SG (2001) Teaching modeling in management science. *INFORMS Trans. Ed.* 1(2):62–67.

Powell SG, Baker KR (2019) *Business Analytics: The Art of Modeling with Spreadsheets*, 5th ed. (Wiley, Hoboken, NJ).

Ragsdale C (2017) *Spreadsheet Modeling & Decision Analysis: A Practical Introduction to Business Analytics*, 8th ed. (Cengage Learning, Boston).

Ragsdale CT (2001) Teaching management science with spreadsheets: From decision models to decision support. *INFORMS Trans. Ed.* 1(2):68–74.

Rodrigues B (2023) Building reproducible analytical pipelines with R. Accessed July 28, 2023, https://raps-with-r.dev/.

Roy A; Cookiecutter Community (2012) Cookiecutter: Better project templates. Accessed July 28, 2023.

Rule A, Birmingham A, Zuniga C, Altintas I, Huang SC, Knight R, Moshiri N, et al. (2018) Ten simple rules for reproducible research in Jupyter notebooks. Preprint, submitted October 13, https://arxiv.org/abs/1810.08055.

Savage S (2001) Blitzograms—interactive histograms. *INFORMS Trans. Ed.* 1(2):77–87.

Savage SL (2012) *The Flaw of Averages: Why We Underestimate Risk in the Face of Uncertainty*, 1st ed. (Wiley, Hoboken, NJ).

Treadway A (2023) *Software Engineering for Data Scientists* (Manning, Shelter Island, NY).

Ufford M, Pacer M, Seal M, Kelley K (2019) Beyond interactive: Notebook innovation at Netflix. Accessed July 28, 2023, https://netflixtechblog.com/notebook-innovation-591ee3221233.

Virtanen P, Gommers R, Oliphant TE, Haberland M, Reddy T, Cournapeau D, Burovski E, et al. (2020) SciPy 1.0: Fundamental algorithms for scientific computing in Python. *Nature Methods* 17(3):261–272.

Walls P (2023) Mathematical Python. Accessed July 28, 2023, https://github.com/patrickwalls/mathematicalpython.

Wastl E (2023) Advent of code. Accessed December 6, 2023, https://adventofcode.com/2023/about.

Wickham H (2014) Tidy data. *J. Statist. Software* 59(10):1–23.

Winston WL, Albright SC (2018) *Practical Management Science*, 6th ed. (Cengage Learning, Boston).