Case Studies and Modules for Data Science Instruction

Python Programming in Digital Agriculture

Sneha Jha[1,*], James V. Krogmeier[2], Dennis R. Buckmaster[1], Andrew D. Balmos[2]

Highlights

- The module introduces Python programming with applications.
- Ample examples in well-documented workbooks demonstrate each learning objective in detail.
- Publicly available historical yield and weather datasets are used for data handling examples, making the module reproducible and open source.

Abstract. *Python has gained immense popularity in academia and industry, owing to its readability, interoperability, and clear code block distinctions. Its open-source nature has fostered a vibrant online community contributing to the development of Python libraries. In academia, Python has become the dominant choice for computer science instruction due to its versatility and wide-ranging applications, including web development, machine learning, and scripting in various software tools. However, due to the abundance of Python learning resources online, beginners often face information overload, making it challenging to navigate the vast array of tutorials, courses, and blogs. This abundance of resources may leave instructors or self-learners struggling to find a cohesive and simple way to master Python, particularly outside the domain of computer science. To address this knowledge gap, our instructional module (nominally 8-10 hours of instruction) offers a streamlined approach for both instructors and students, especially aiming towards Python's prowess in handling agricultural data analytics. It provides clear learning objectives in Python from setup and installation techniques, programming, recommending relevant resources, and assessing learning with practical exercises to assist throughout learning. By focusing on good instructional practices and presenting a set of real-world data analysis problems tailored to agricultural data applications, this module aims to improve the efficiency and effectiveness of teaching programming skills in Python.*

Keywords. *Data Analytics, Digital Agriculture, GitHub, Public Datasets, Python programming language, VS Code.*

MOTIVATION AND OBJECTIVES

Python is one of the fastest-growing programming languages in the world, distinguishing itself for software applications in industry and academia. The programming of software can be generally ascribed to one of three purposes (i) to run simulations or solve complex problems, (ii) to wrangle and analyze data to generate insights (which includes AI and ML), and (iii) to control things (outputs) based on inputs (such as sensors) in automated manners. Python is universal enough that it has been deployed in all three realms. Python has also become the most popular choice as the primary coding language taught in university computer science curricula (Guo, 2014; Siegfried et al., 2021). The written structure of Python code is easy to learn and produces concise programs with common English keywords and line breaks to differentiate its code blocks. This makes it easier to read and share. In addition, Python is open source which has helped it to build a large presence in the online community helping to spur Python development and dynamic evolution of the language through the introduction of numerous libraries of specialized code.

Python is being used as the backend in many popular web applications (Steponaitis, 2020), to develop websites, in machine learning and artificial intelligence, and as a scripting language in software such as ArcGIS, QGIS, FreeCAD, SPSS, and MySQL workbench to automate repetitive tasks. These examples of Python applications alone make it one of the most useful tools for students or scholars, more so if pursuing digital agriculture. The goals of digital agriculture research and education are to enable real-time analysis and automated decisions based on precision agriculture data. To fulfil these digital agricultural requirements, scholars,

[1] Department of Agricultural and Biological Engineering, Purdue University, West Lafayette, Indiana, USA.
[2] Elmore Family School of Electrical and Computer Engineering, Purdue University, West Lafayette, Indiana, USA.
[*] Correspondence: jha16@purdue.edu

Submitted for review on 27 January 2024 as manuscript number EOPD 15961; approved for publication as a Teaching Module by Associate Editor Dr. Deepak Keshwani and Community Editor Dr. Monica Gray of the Education, Outreach, & Professional Development Community of ASABE on 6 June 2024.

engineers, and developers are expected to work with modern software tools like ArcGIS, MySQL, QGIS, Pix4D etc., and Python is the choice of scripting language for these tools.

A Google search on "learn Python" brings up more than one billion results, most of which are for beginners. These include tutorials, blogs, and courses from reputable companies, universities, websites, and programmers. However, this vast community and the myriad of online Python tutorials can become overwhelming for beginners. With so much already available tutorial material, one should wonder about the wisdom of creating more.

The primary objective of this instructional module is to concisely share good practices of instruction and provide example problems for instructors and students that enable efficient learning of Python with a focus on agricultural applications. In doing so, this teaching module formulates learning goals in Python programming, identifies good resources for Python programming, and provides exercises to evaluate learning. The module developed takes nominally 8-10 hours of instructional (class) time and specifically targets novices in the Python language.

This paper is organized as follows. In section 2 of the paper, we describe the tools used during lesson delivery in detail and list the broad learning outcomes from the course modules. Section 3 describes in detail the Python Demonstration Workbooks used in the lecture and the Practice Workbooks used during the laboratory sessions. The sub-sections in section 3, are organized to follow a gradual transition from basic to advanced programming skills with the help of simple and practical digital agriculture data sets. The reader is encouraged to follow along with the workbooks provided in the GitHub repository associated with this paper (https://github.com/oats-center/PythonInDigitalAg) for a hands-on learning experience. Finally, in section 4 we conclude the paper with a general discussion.

MATERIALS AND METHODS

Since the course module is designed to introduce Python programming for digital agricultural students with little or no previous experience in coding, the resources include aspects of setting up a simple coding environment in VS Code, workspace management and code sharing on GitHub. Python codes are written in Jupyter Notebooks and are publicly shared on GitHub. Mybinder.org is used to demonstrate programming examples during class. It facilitates students to follow the lectures without the need to install Python on their machines. MyBinder (Jupyter et al., 2018) is an open-source online service used to run Jupyter notebooks from GitHub repositories.

Overview of the resources used to create and publish the developed course on GitHub:

i. Python's official documentation (Python Software Foundation, 2023), The Examples book (The Data Mine, 2020), a Python Tutorial by Guido and the Python development team (Guido van Rossum, 2018) and published journals referenced in this paper are shared for reading in the Resources folder.
ii. Python workbooks are uploaded and shared from GitHub (Krogmeier et al., 2024) in the form of Jupyter notebooks. Binder links are also provided in the repository for easy web access.
iii. Online resources used during the class lectures are also referenced in the Jupyter Notebooks and relevant sections of this paper.
iv. The United States Department of Agriculture National Agricultural Statistics Service (USDA-NASS) historical wheat and corn yield data (U.S. National Agricultural Statistics Service NASS., 1998) and applied climate information system (ACIS) weather data (ACIS, 2017) used in exercises on scraping, wrangling, analyzing and visualizing data in Python are provided in the Data folder of the repository.

LEARNING OBJECTIVES OF THE MODULE

Upon successful completion of this 8 to 10-hour teaching module, readers will be able to

i. Understand and navigate through the integrated development environment (IDE) setup.
ii. Install statistical, scientific, data processing and visualization packages in Python and apply them to agricultural data sets.
iii. Write Python code to analyze and visualize agricultural data and publish results online.
iv. Manage a version control system and use code sharing for collaborations.
v. Create more Python tutorials and customize them to suit specific requirements (see Appendix).

LESSON DELIVERY

The course module has 4 Demonstration Workbooks and 2 Practice Workbooks. Supporting content is provided in YouTube recordings and the Workbooks are publicly available on GitHub (Krogmeier et al., 2024). Binder custom environments were used during lectures to demonstrate the workbooks in class. The students were encouraged to follow along with the instructor and were free to use either the Binder method or run the workbooks on their machines. Students were guided through a tutorial to download the course repository from GitHub onto their computers. Demonstration Workbooks were used during the lectures for visualization and student participation. The Demonstration Workbooks contain detailed examples of topics covered in this course module. Two Practice Workbooks A and B were used during the laboratory sessions and are designed to help the students work on programming exercises and assess their Python skills. The solutions to the

exercises in Practice Workbooks are available in the course repository on GitHub. Datasets used in the workbooks are also published in the course repository.

To introduce novices to Python programming and ensure enough familiarity for self-learning using public resources, the workbooks are assembled to guide learners from basics in the Python language and IDE usage to applications of Python libraries to analyze agricultural data.

The objectives of each Demonstration workbook are:

- Demonstration Workbook 1: Introduce data types, syntax, simple functions, and control commands.
- Demonstration Workbook 2: Introduce modules and packages for data analyses and creating appropriate visualizations.
- Demonstration Workbook 3: Familiarize learners with the capability of *Pandas* to automate data handling by scraping weather data, preprocessing it for wrangling, and calculating data derivatives.
- Demonstration Workbook 4: Introduce simulation via a children's board game and combine it with analyzing game performance statistics.

The Practice Workbooks A and B are designed to guide through the process of (i) setting up a working environment for programming in Python language, (ii) illustrating the syntax and good commenting practices in programming, and (iii) providing hands-on coding experience and exercises on the topics discussed during lectures. The two Practice Workbooks are accompanied by their solutions and the topics covered in these workbooks are:

- Practice Workbook A: Set up the VS Code IDE, install libraries, and exercises based on topics from Demonstration Workbooks 1 and 2 such as defining variables, formulating logical conditions, and writing functions.
- Practice Workbook B: Examples of leveraging advanced *matplotlib* functions to customize figures for complex graphs and exercises on visualizing datasets from Demonstration Workbooks 3 and 4.

DEMONSTRATION WORKBOOK 1: INTRODUCTION TO TOOLS AND FUNDAMENTALS OF PROGRAMMING IN PYTHON

The rapid growth in Python's user community is attributed to its readability, a fundamental aspect Guido van Rossum prioritized during its development in 1989. An additional motivation was the visionary idea of open-source release. As a result, the language has continuously evolved since 1989, progressing to its third version and accommodating hundreds of third-party libraries encompassing areas such as statistics, mathematics, data processing and visualization, geospatial analysis, machine learning etc. Python's most recent stable release, version 3.12, became available in August 2023. The course module associated with this paper is compatible with all versions of Python 3.

Syntax

The fundamental principles observed when writing in any programming language convey the compiler's comprehension of the structure of the elementary statements, the mechanism for linking these statements to create a logical flow, and the various methods of data input and output (Raphael, 1966). Python, like any other programming language, adheres to its own set of rules governing these fundamental processes. The first Demonstration Workbook discusses these fundamental processes in Python's context and provides example codes to illustrate their proper execution.

Programs have unique syntax, which consists of a set of rules for writing in a programming language. This syntax governs every aspect of writing a program such as the initialization of variables, differentiation between code blocks, construction of ordered control statements, creation of functions, usage of functions, and so forth. Maintaining proper syntax is essential for reading and executing a code using a Python interpreter/compiler. During the initial stage of program execution, known as the parsing stage, the Python interpreter examines the syntax and halts execution if it detects any error. These errors are called "*Syntax Errors*". Therefore, fluency in Python syntax is crucial. Figure 1 is used to illustrate the syntax which is discussed in the following paragraph.

Python, like other programming languages, defines its constructs using special characters such as *"()", "=", "%", ":", "{}", "[]"*. The special character syntax can be called the grammar of Python and is discussed throughout the paper. Another important aspect of syntax in Python programming language is the spacing/indentation between code blocks. Other programming languages such as *C* and *JavaScript* use semicolons to signal the end of a line or curly braces to indicate the end of a code block whereas Python uses indentations. A few examples are: (i) the lines after declaring a for-loop also called the body of the for-loop require a single indentation, the loop ends when the indentation is terminated (ii) a function should have a single indentation before each line in the code block written to define its operation also called the body of the function, the definition ends when the indentation is terminated, also illustrated in Figure1. The importance of these indentations has triggered modern IDEs to implement predictive typing for indentations at appropriate positions based on Python's syntax. The indentations are enforced strictly, and Python will halt and print an "*Indentation Error*" message during runtime if such an error is detected. Comments in Python are lines explaining the codes written generally above or to the right of the code and have a unique syntax. Python uses a "#" symbol before the start of a single-line comment and three pairs of quotes to enclose a multiple-line comment illustrated in Figure 1. Incorporating detailed comments within a script enhances its comprehensibility and shareability. Additionally, color coding, a feature often dependent on IDEs, proves valuable for code readability. Users have the flexibility to select their preferred color themes without impacting the interpretation of the code. Discussions 1 in Demonstration Workbook 1 explain the basic programming syntax, printing output and writing comments with examples.

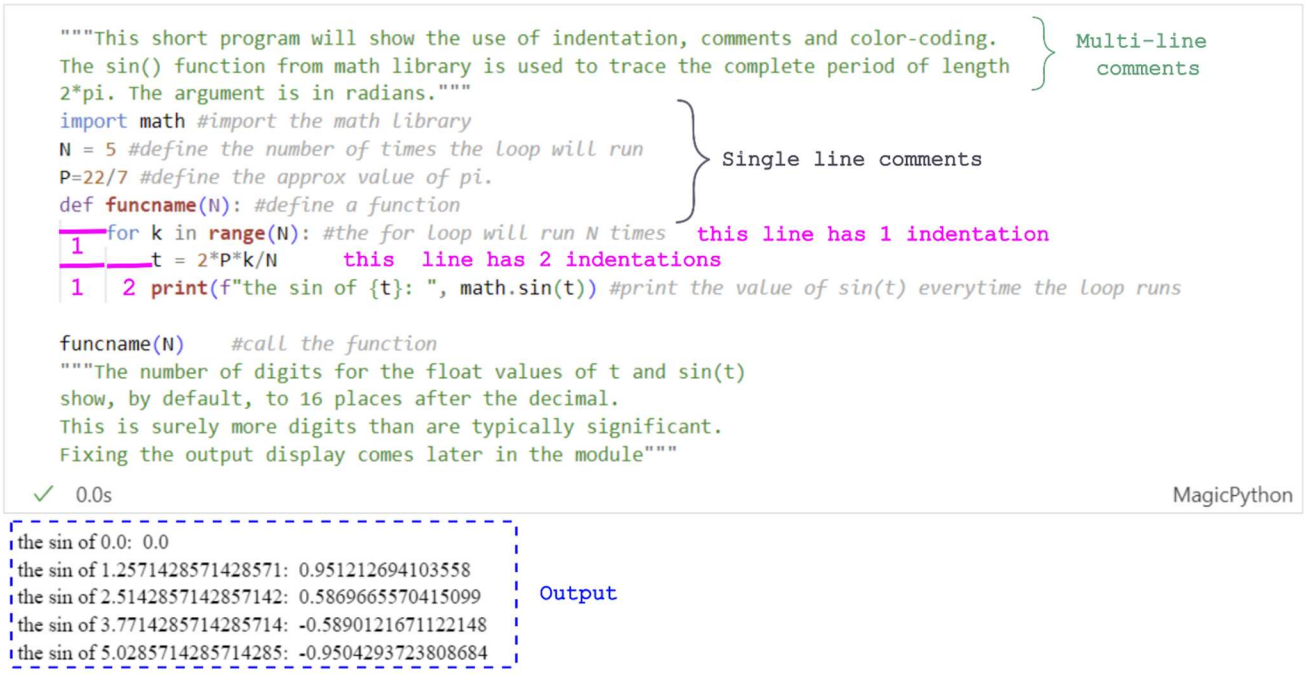


Figure 1. This figure illustrates the use and effects of indentation, comments and color coding in editor using a sample code block.

Data Type and its Methods

Data in Python has three features (i) identity, (ii) type, and (iii) value. All data in Python is interpreted as memory objects by its compiler and called simply objects in general. The identity of a data object is fixed to the address of the memory used to store it. The type of the data object determines the format of values it can store such as integer, string, float etc. The data type also determines the operations that can be implemented using the data. The value of the data object is declared by the programmer or detected during compilation. Python interpreter detects the object's type from its values and allocates the required identity and space to the object in its memory.

A Python variable is a human-defined reference name for the data objects. In Python, variable declarations are dynamic, meaning a data object can be initialized simply by assigning a value to a variable name, anywhere in the program before the variable is used. A few examples of variable declaration are illustrated in Figure 2. There are some important rules regarding variable declaration. For example:

- A variable's name may start with an underscore, but it must not contain special characters such as *"!, @, #, $, %, ^, &, * "*.
- Variable names are case-sensitive.
- Python syntax keywords cannot be used as variable names, e.g., *raise*, *global*, V, *True*, *while*, *None*.

```
"""Variable declaration is dynamic and does not require
data type declaration. The Python interpreter automatically
assigns the correct datatype to the variable."""
x = 4 #an integer example
y = 20.5 #a float example
z = "Python is for all!" #a string example
p = 4+5j #a complex number example
pi = 22/7 #approximate pi value is a float
print('The value of x is:',x)
print(f'The value of x is {x} and its data type is:',type(x))
print(f'The value of y is {y} and its data type is:',type(y))
print(f'The value of z is {z} and its data type is:',type(z))
print(f'The value of p is {p} and its data type is:',type(p))
print(f'The value of pi is {pi} and its data type is:',type(pi))
```

✓ 0.0s MagicPython

```
The value of x is: 4
The value of x is 4 and its data type is: <class 'int'>
The value of y is 20.5 and its data type is: <class 'float'>
The value of z is Python is for all! and its data type is: <class 'str'>
The value of p is (4+5j) and its data type is: <class 'complex'>
The value of pi is 3.142857142857143 and its data type is: <class 'float'>
```

Output

Figure 2. This code snippet shows variable declaration.

Data objects can also be tuples, sets, lists, and dictionary data types. These are also called collection data types. They act like containers or structures which consist of multiple data objects (The Python tutorial, 2023). The data objects in collection data types are often indexed, ordered and mutable. Python's data objects can be immutable such as the single-valued data types (numbers, float, string) and tuples, or mutable such as the collection data types (sets, lists and dictionaries). Unlike immutable data objects, the number of values in a mutable data object can be changed. For example, a tuple cannot have more than two values whereas the number of values in lists, dictionaries, and sets can be changed anytime. Moreover, collection data types permit the mixing of various data types within a single data structure.

Each collection data type has a well-defined structure and characteristics. Structurally, tuples accept two variables within parenthesis separated by a comma, lists accept any number of variables within square brackets separated by a comma, sets accept any number of unique values in curly braces separated by a comma, and dictionaries accept any number of "*key: value*" pairs in curly braces separated by commas. These collection data types differ in their characteristics too, for example, set objects are unordered and unindexed which makes them immutable. Whereas a list element can undergo most kinds of mutations because list objects are indexed and ordered collections of variables. The structure and characteristics of the collection datatypes are illustrated with examples in Discussion 2 of Demonstration Workbook 1.

Python uses the data type to check the methods that can be implemented using these objects. Methods in Python are built-in codes which perform a specific task and are used by referencing them when required. Methods are unique to the data type of an object such as "*str.split()"* (splits a string object) and *"str.replace* (replaces a string object), are methods of the string object, "*list.append()"* (appends objects to a list) and *"list.remove()"* (removes objects from the list) are methods of the list object. These methods can only be used with their data types. Examples of methods are worked out in Discussion 2 of Demonstration Workbook 1.

Operators

The different data types facilitate important mathematical, logical and data manipulation functions. These functions play a crucial role in data manipulation tasks and are often dependent on logical statements or expressions with appropriate operators.

Operators in Python are classified into four main categories based on their primary functionality:

1. assignment operators (=, +=, -=, =, /=, *=)
2. arithmetic operators *(+, -, /, %, *)*
3.comparison operators *(==,!=, <, <=, >, >=)* and
4. logical operators (*and, or, not*).

These can be used in different combinations to perform more functions which are illustrated in Discussion 3 of Demonstration Workbook 1 with suitable examples.

Tools for Flow of Control in Python Programming

Python interprets codes sequentially from top to bottom. This is the natural direction of controlling the data manipulation in the program. The variables must be declared before they are used in operations and operations must be written in the sequence of the logic. Operations use logical statements and expressions to control the sequence of interaction of objects. Pythontutor.com an online code compiler visualization tool (Guo, 2021) was used to explain the flow of code during class demonstrations.

Python programming execution is frequently controlled using compound statements such as *for, while*, and *if*. These are used to create conditional loops (The Python tutorial, 2023). The *for* and *while* compound statements are examples of iterations or loops in Python and are used for implementing single or multiple-condition statements. This iterative calculation is the crux of using programming languages in engineering. The *for* and *While* loops have different syntax for their definition. Structurally, the *for loop* begins with a definition of the range of the loop using "***for*** *iterator* ***in*** *iterable-range:*" syntax. The code block succeeding the *for loop* definition executes operations repeatedly until the loop range is completed. This code block is called the body of the *for loop*. The body of the *for loop* starts from the next line and must be written with an extra indent from the definition of the loop. Functionally, the *for loop* in Python can iterate over the range of any data type, without defining iteration steps or the halting condition. It halts and moves on to the next statement when it completes looping over the whole range without any error. The *for loop* is generally used to initiate iteration within a range of values. Creating a list of elements/variables or printing elements of a list is a simple and popular way of using *for loops*. More examples of for loops are illustrated in Discussion 4 in Demonstration Workbook 1.

The *while loop* on the other hand checks a condition and iterates while the condition is true. Structurally, the *while loop* is defined using a "***while*** *(condition):*" syntax. The body of the *while loop* is defined in the same way as for the *for loop*. *While loop* may be defined with an ***else***, ***break*** or ***continue*** clause. These clauses are written sequentially in the execution flow. Functionally, the *while loops* are used when the number of iterations required is unknown or infinite. Its built-in helper clauses ***else, break,*** and ***continue****,* are leveraged to halt the *while* loop to accommodate more conditions. The *for* and *while loops* can sometimes be used interchangeably depending on the situation and programmer. The usage of the *for* and *while* loop is illustrated with examples in Discussion 4 of the Demonstration Workbook 1.

The *if-else* compound statement works as a conditional expression. Structurally, the *if* expression has "***if*** (condition):" syntax. It is followed by a code block which is written to execute operations sequentially and is called the body of the *if* condition. The body is written in the next line of the definition and at an extra indentation from the definition line. The if condition is accompanied frequently by an else clause written as ***elif***. Functionally, the *if* can execute the following expressions if the condition is true based on its criteria or move on to the next line in the code. The *if* statement uses the *else* and *else-if (elif)* clause to include multiple follow-up criteria in the code. The *if-else* statement is popular in sorting, classifying, and search algorithms. Examples in Discussion 4 of Demonstration Workbook 1 illustrate the syntax and implement simple operations using *if-else* conditional statements.

Functions are included in the programming tools to control the flow of the program; however, they are different from the iterators or conditional statements discussed above. Functions are code blocks which perform a set of defined actions on variables when referenced. This act of referencing a function by its name is termed as "calling a function or function-call". Structurally, functions are written with a definition and a body. The syntax for the definitions is ***"def*** *function-name (arguments):".* A function name should be unique and follow the same rules as a variable declaration. The arguments are the variables required for the execution of the function body. The body of the function starts from the next line at an extra indentation from the function definition line. The function ends with a ***return*** statement, this statement may return a value or *None.* A function which does not return any parameter is called a routine. Functions work in two steps. The functions first need to be defined and then called. A function can be called anywhere in the code by referencing the function name and passing values to the argument required by the function definition. On execution, the variable assigned to the function-call stores the values returned by the function. There are many built-in functions in Python such as print (), max (), min (), list (), type () etc. and a complete list of Python's built-in functions can be found in its official documentation (Python Software Foundation, 2023). In Discussion 5 in Demonstration Workbook 1, a function is written to compute the square root of a number. Discussion 5 also illustrates the correct usage of functions in concise and clean programming.

The Python programming syntax for variables and control tools for the execution flow are important and programmers should be comfortable in using these by the end of this workbook.

DEMONSTRATION WORKBOOK 2: MORE PYTHON TOOLS FOR PROGRAMMING AND INTRODUCTION TO AG DATA ANALYSIS

The second Demonstration Workbook of this course module introduces the Python packages and guides the learners towards the application of the programming basics from prior discussions. This workbook also introduces Python programming techniques for data analysis using example data from agricultural practices in the USA.

Python Modules and Packages

A program in Python which performs a task on compilation is called a script. Functions and routines which are not written in a program can still be called in the program using Python modules. Advanced programmers often store subsections of their code called scripts in separate Python (.py) files. These scripts are then referenced in other programs to avoid creating

a single lengthy program and ease maintenance tasks. These scripts are called modules. Modules can be referenced using the *import* statement and the process is called importing a module. Python has several built-in modules such as *math, datetime, ftplib, glob, itertools, and stat* to name a few among a hundred others. (PythonModuleIndex, 2023).

Table 1. List of a few Python's built-in modules used frequently.

Python built-in modules	Description
math	Mathematical functions
datetime	Basic date-time data handling
ftplib	File transfer protocol (FTP) client management
glob	Unix shell style path name pattern expansion
itertools	Iterators for efficient looping
statistics	Mathematical statistical functions

The built-in modules may need to be explicitly imported into the script using the ***"import** name"* syntax. The code to import a module should be written in the script once, before its first usage. Once imported, the *"name.function"* syntax is used to call the module's functions. For instance, to use the value of *pi* from the *math* module, a script should first import the math module and then call the pi variable as "*math.pi*" as shown in Figure 3.

```
import math # This will import math library
P = math.pi #assigning the value of pi to P
print("the value of Pi from math library: ",P)
power_result = math.pow(2, 3)      # Exponentiation: 8.0
print("2 raised to the power 3 = ",power_result)
ceil_result = math.ceil(3.2)       # Ceiling function: 4.0
print("rounding up of 3.2 = ",ceil_result)
floor_result = math.floor(3.9)     # Floor function: 3.0
print("rounding down of 3.9 = ",floor_result)
factorial_result = math.factorial(5)  # Factorial: 120
print("factorial of 5 = ",factorial_result)
```

✓ 0.0s MagicPython

```
the value of Pi from math library:  3.141592653589793
2 raised to the power 3 =  8.0
rounding up of 3.2 =  4
rounding down of 3.9 =  3
factorial of 5 =  120
```

Output

Figure 3. Illustration of the implementation of a few methods from the math library taken from Demonstration Workbook 2.

As an open-source language, Python has a large active community of developers who create single modules or combine multiple related modules, which provide a certain functionality. These groups of modules are called third-party packages such as *NumPy, SciPy, rasterio, requests, seaborn, Pandas,* and *matplotlib*. These third-party Python packages are a library of functions, routines and variables written in separate Python modules, documentation, and installation files packaged into a single installable file. Most of these packages are distributed with open licenses, are managed by the Python software foundation and are available on the Python Package Index (PyPI) website.

Table 2. List of a few Pythons' third-party packages used frequently.

Python third-party libraries	Short description of these libraries
NumPy	Scientific computing, mathematical models, array objects.
SciPy	Math algorithms built on NumPy for science and engineering.
Pandas	DataFrame data structure, data analysis tools.
GeoPandas	Data analysis operations like Pandas on geospatial data.
requests	Hypertext transfer protocol (HTTP) library.
rasterio	Reading, writing, and processing geographical raster files.
Matplotlib, seaborn	Creates static, animated, and interactive visualization.

There are several ways to download and install packages even from PyPI. The packages hosted by websites like PyPI use package installers/managers to allow users a single-step package installation. We have used the package installer for Python (pip) (The pip developers, 2008) with PyPI to illustrate the working of package managers in this course module. The Installation and Setup file of this course module guides the users through each step of installing and using pip with PyPI.

The installed Python packages can be used in a script by importing them using the "***from** package **import** module"* statement. The import statement must be called once before its first usage. Another popular method of importing is to use a

short alias, "***import*** *package* ***as*** *alias*" while importing modules from the package. This makes it easy to use the function/routine library of the packages. The alias method decreases the time spent repeatedly writing the full package name each time a method of the package is called. There are several ways of importing modules which are illustrated here in Figure 4 and in Discussion 1 of the Demonstration Workbook2.

Discussion 2 in the Demonstration Workbook 2 incorporates a few modules and third-party packages/libraries such as *math* (The Python Standard Library, 2023), *NumPy* (Harris et al., 2020), and *matplotlib* (Hunter, 2007) in the discussions on importing packages/libraries.

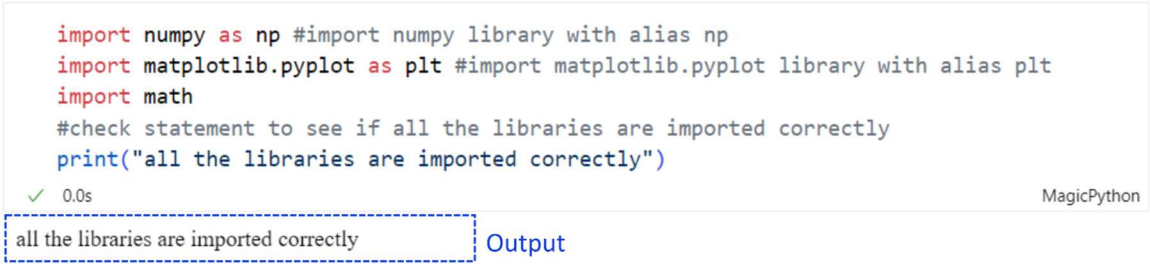


Figure 4. Illustrates some of the different ways of importing packages/libraries and modules in Python.

The *math* module of Python facilitates basic mathematical operations using simple numerical data types. *NumPy* is a library of mathematical and scientific functions. It is equipped to handle large datasets efficiently. *Matplotlib* is a Python graphing library that helps to program 2-dimensional/3-dimensional plots, interactive plots, and animated plots.

The examples to illustrate the usage of important packages such as *NumPy, matplotlib and Pandas* along with a few other modules are carefully curated in the Demonstration Workbooks 2, 3 and 4 of the course modules. The corresponding discussions are mentioned in this paper under proper sections for reference.

Data Input and Output

After familiarization with Pythons' programming syntax and its libraries, the next crucial step is understanding its data communication approach. Python receives information through various data input actions and presents results or error messages as outputs. These outputs are typically displayed in the designated output display space. It is important to note here that they are only available for the current editing session. Once the programming window is closed, the input variables and output variables are cleared. Therefore, it becomes essential to save processed data into files for future reference.

Data input methods in a Python program can be categorized into three types: (i) input from a user, (ii)input from a file saved in the local computer and (iii) input from scraping data from a remote database on the internet. Similarly, data output methods have three categories (i) displaying output in a designated window, (ii) saving output to a file on the local computer, and (iii) saving it on a remote database on the internet.

Python programs can take user input during runtime and set it to variables. Examples of this data input method are illustrated in Discussion 3 of Demonstration Workbook 2. Python employs the *"open()"* method to access the locally stored data file from the computer's hard drive. The open method accepts the path of the saved file as its argument and creates a file object to read its content. The path of a file is the complete address of the file in the computer's file system. There are two ways to define the path of a file (i) absolute path: the address of the file from the root folder (generally C drive in the Windows Operating System) in the computer, and (ii) relative path: the address of the file relative to the current folder. The file object can use the absolute or relative path to access the file from the local computer's hard drive. A file object is a special data type, exclusively used for reading text data from a file and writing it to a file. It offers several methods such as "*file.readline", "file.write",* and *"file.close"* among others. The underlying processes of the file object are explained using the "*Limerick"* file from the Data folder of the course module. In the example:

1. A prompt is created for the user to insert the relative path of the Limerick file present in the Data folder.
2. The filename is stored in the "my_file_name" variable using the user input.
3. The *"open()"* method creates a file object and reads the content of the file. A *"print()"* statement is used to print the data from the file to the output terminal.
4. The file object stores the text from the Limerick file, which is a short poem intended to be amusing. An example function is called to reverse the limerick.
5. The reversed limerick is written into a new file in the Data folder.

The examples in Discussion 3 of the Demonstration Workbook 2 illustrate the first two of the listed data input and output methods in Python. We will discuss the input method for data from remote databases in Demonstration Workbook 3.

Ag Data Analysis I: Introduction to Data Analysis Techniques in Python

Python programming language has all the required tools for comprehensive data analysis. The following discussions in the workbook illustrate the usage of Python's various functions and methods to program the necessary steps for wrangling data before analysis. Discussions 4 - 9 in the Demonstration Workbook 2 illustrate Python methods to (a)parse the public Ag data, (b)preprocess it into clean data, and (c)visualize and compare various crop yield and land use data.

USDA conducts numerous agricultural surveys annually and provides the results and raw data through national agricultural statistics services in the public domain. USDA's Quick Stats database which hosts historical agricultural data and statistics from the United States is one such service. It facilitates downloading current and historical data directly from its website and via application programming interfaces (APIs). We have illustrated the manual download method of Quick Stats data in Figure 5, where the required fields are selected then the "*Get Data*" button is used to download the data. Quick Stats is a valuable resource for researchers, farmers, policymakers, and data scientists interested in agricultural information and analytics. Users can customize their queries and download specific ag data in various time ranges and for different states. These are especially useful for statistical analysis of trends in agricultural yield based on geolocation, crop type, and land use over time.

Most data analysis problems begin with a purpose statement followed by data collection, processing of raw data to a standard format, analysis, and visualization. Discussions 4- 9 in Demonstration Workbook 2 follow this sequence and present a few example case studies using wheat, corn, and land use data from the Quick Stat database to demonstrate the scope of Python in simplifying and accelerating the processing, analysis, and visualization of agricultural data. It's important to note that the approach used to demonstrate techniques in Python for agricultural data analysis problems is not the sole or necessarily the most optimal approach. We have prioritized illustrating the usage of various Python methods and functions instead.

To illustrate the first example of data analysis, corn yield data from 1866 to 2021 has been downloaded using the variables highlighted in Figure 5 and made available in the course module folder on GitHub. The downloaded corn yield data is saved in the "US-National-CornYields.csv" file in the Data folder of the course module. The "*file.open ()"* method uses the relative path of the downloaded data as the argument and creates a file object to read this comma separated value (CSV) file. The *"print()"* statement is used to display the contents of the file in the output terminal.

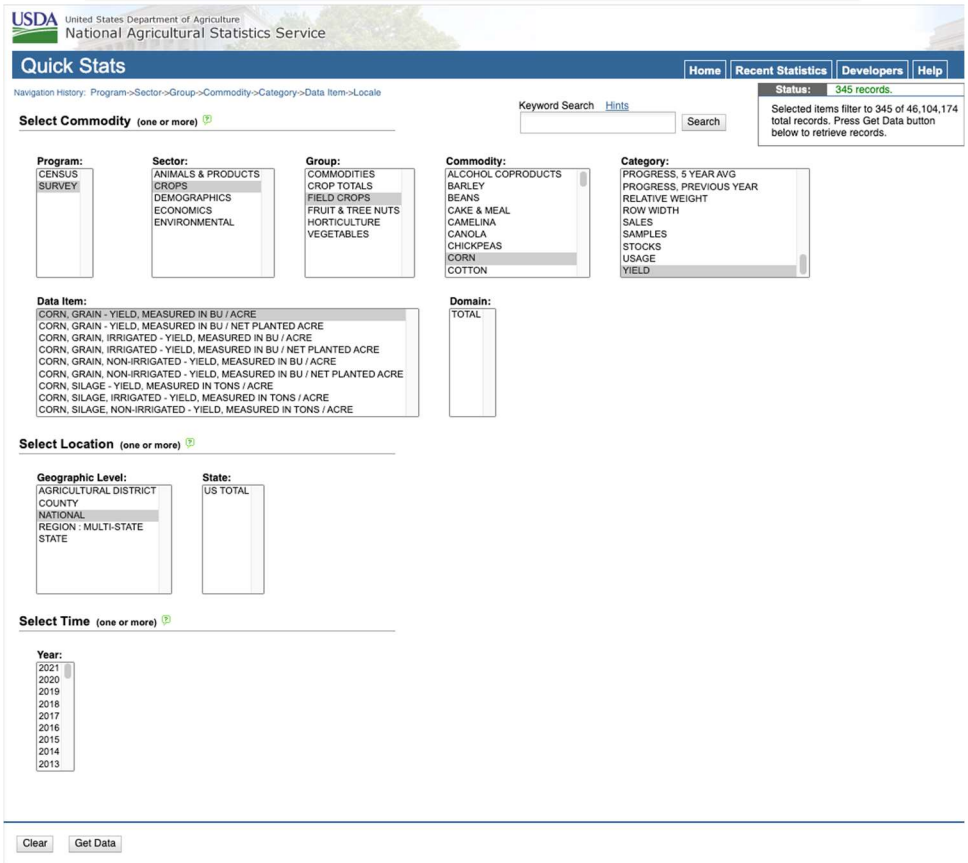


Figure 5. The shaded parameters were selected to download the surveyed historical national corn yield data in bushels/acre (bu/ac) from 1866 to 2021.

Python's string manipulation methods treat each character of a string as an indexed value in the string object. The examples in Discussion 4 and 5 for Demonstration Workbook 2 take advantage of this feature and parse the input strings of the corn data as single-character string objects and split them to create useful data columns using *string.replace()* and *string.split()* methods. The *string.replace()* method is used to replace each line break with a space before splitting the data at each space by using the *string.split()* method. Using these methods *NumPy* 1D arrays containing yield (represented as floating point data) and year (represented as integer data) values are created. It should be noted here that *NumPy* accepts

data as N-dimensional array (ndarray) objects. The ndarray allows *NumPy* routines and functions to speed up operations. *NumPy*'s built-in statistical functions are applied to the yield array. The central tendency of yield values is computed using *NumPy*'s built-in functions *min()* for the minimum yield, *max()* for the maximum yield, *std()* for the standard deviation of the historical yield values, and *mean()* for the average yield values over the years. The line plot for the historical yield of corn was generated using modules from the *matplotlib* graphing library. Equivalent to ndarray from *NumPy*, matplotlib uses the figure as its object data type for its operations. A 'figure' object is assigned a designated frame/space with default dimensions in which the graph is displayed in the output. *Matplotlib* allows modifying all the graph features such as title, axis label, axis range, legend, and color of the graph among others to customize a figure. In Discussion 6 of the Demonstration Workbook 2, the corn and wheat yield data are used to illustrate matplotlib's multiple plot functionality. A multiple plot allows more than one graph drawn in a single figure. In Discussion 6, a figure object is used to draw the historical wheat yield in green and the historical corn yield in blue as shown in Figure 6. The graph legend, axis labels and titles are customized using matplotlib methods. The Y-axis shows the yield values in bushels per acre and the X-axis ticks represent the years in which the yield data was collected. It should be noted here that the wheat data is taken from Discussion 2 of Demonstration Workbook 2 illustrating the usage of *NumPy*.

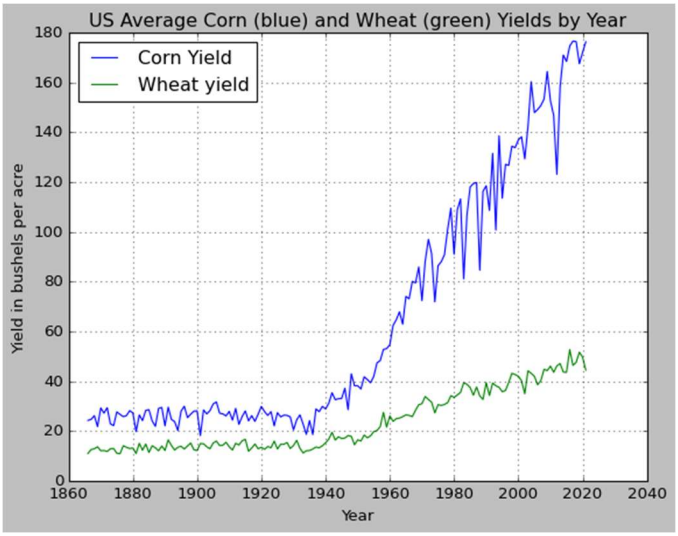


Figure 6. An illustration of multiple graphs in a single figure pane output in matplotlib.

In yet another example, an analysis of the values of non-irrigated land in selected states of the USA from 1997 to 2021 is presented in Discussions 7 and 8 of the Demonstration Workbook 2. The land use data used in this example is available in "US-LandValue-by-State-and-Year.csv" in the Data folder of the course repository. Discussion 7 introduces more string manipulation methods to parse the complex raw data of land usage in the US. Discussion 8 illustrates a function (*LandValue-ByState*) which (i) parses the raw land use data and extracts state, year, and land value information using string manipulation techniques, (ii) selects land values of three American states, and (iii) saves them in NumPy arrays. It should be noted that this function facilitates selecting and processing the raw land use data for any three US states simply by passing them as arguments to the function. The arrays returned by the "*LandValueByState*" function are used to explore the central tendency of land values in the three states of Arkansas, Colorado, and Kansas from 1997 to 2021 using NumPy's statistical functions. These arrays are also used to create a figure to compare the average values of non-irrigated lands in these three states using matplotlib.

At the end of the Demonstration Workbook 2, learners should be able to create variables, import Python packages, call functions from the package libraries, and write functions to create basic data analysis operations.

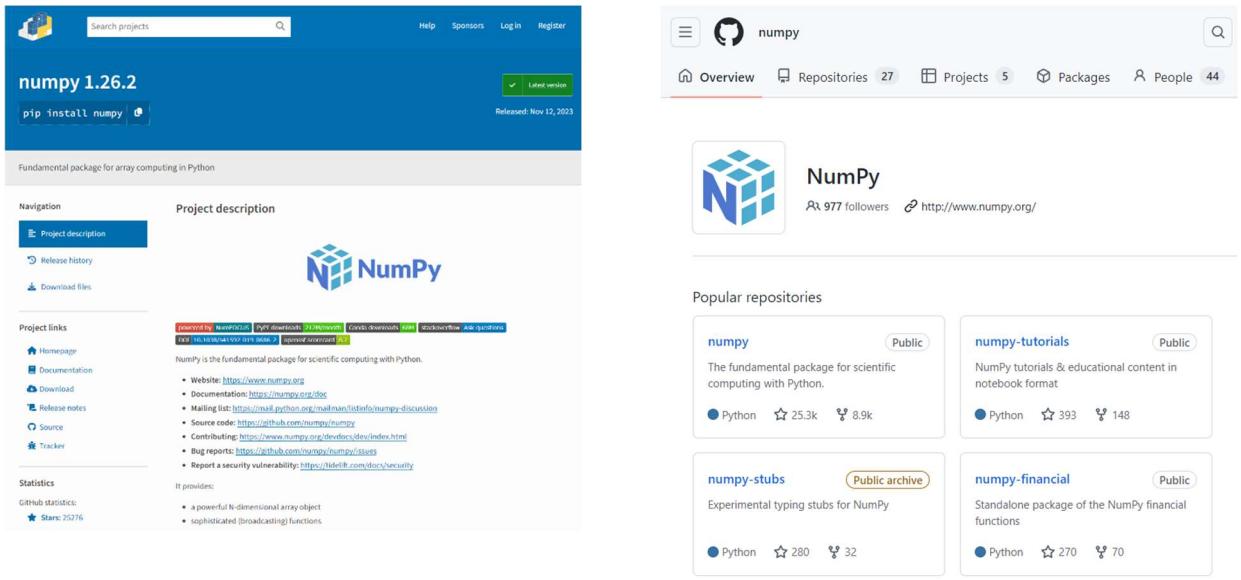
PRACTICE WORKBOOK A: WRITE YOUR FIRST CODE BUT FIRST SETUP

There are several ways to install and run Python on a computer depending on the operating system (OS). Python can be downloaded on any computer and used with the command prompt as the editor to write and compile programming. However, this method uses some advanced understanding of directory structures in different OS. To accommodate novice learners, the integrated development environment (IDE) method is used in this course. Current IDE platforms such as Anaconda, Visual Studio Code (VS Code), PyCharm etc. make it easy to set up Python programming environments. We will illustrate the Python setup in Visual Studio Code, which is easy to install, manage and compatible with both Mac OS, Linux, and Windows OS. VS Code handles all Python's software requirements without any external installation requirements. It also installs Jupyter Notebooks, which are used extensively in this course.

The steps required to install the VS Code software and set up a GitHub profile are described in the shared educational material in the file "InstallationandSetupGuide.docx". This file is available in the course repository on GitHub. On

completing the installation of the VS Code IDE and creating a GitHub user profile, the course module from the course GitHub profile can be downloaded onto the local computers for offline access.

An essential aspect of managing Python on a machine involves familiarity with the Python package installers. There are several package installers such as conda, Python install package (pip), yum, EasyInstall etc. For this module, we have opted to utilize pip (The pip developers, 2008). Pip is recognized as the standard package manager and is typically bundled with Python installation files. Pip downloads and installs the Python packages in the Python directory on the local computer with a single user command *"pip install package name"*. The package installer pip checks the version of Python installed on the local machines and installs the compatible version of the packages in the proper Python directory. A comprehensive list of third-party packages can be accessed on the PyPI website or its GitHub repository (Python Packaging Authority, 2011). Figure 7 (a) and (b) illustrate the source page for downloading NumPy from PyPI and GitHub respectively. For beginners, installing these packages is both convenient and secure through the terminal in VS Code. This approach allows VS Code to integrate and make it available in the VS Code environment.



(a) **(b)**

Figure 7. An example page for downloading NumPy from (a) PyPI and (b) GitHub.

Tasks in Practice Workbook A guides users through the installation, setup, and assessment tasks. The assessment tasks are based on topics which are discussed in the Demonstration workbooks 1 and 2.

The solutions for the tasks in Practice Workbook A are available in the module repository on GitHub.

Upon successful completion of Practice Workbook A, learners will be able to:

i. Navigate through GitHub to create, fork, and download repositories.
ii. Set up a VS Code workspace on a local machine and install Python. Use pip to install third-party libraries.
iii. Understand Python's programming syntax.
iv. Write conditional statements, iterators, and simple functions.
v. Read data files from files on the local machine and save data in files.
vi. Analyze and plot public agricultural survey data using Python's basic *NumPy* and matplotlib packages.

DEMONSTRATION WORKBOOK 3: PYTHON TOOLS FOR AUTOMATING AG DATA ANALYSIS

As we have progressed in this course, we will find advanced methods for performing data analysis tasks which have been discussed previously in the Demonstration Workbooks 1 and 2. These new methods are important because they allow us to write shorter programs which do the same job in less time and decrease effort for users. The Demonstration Workbook 3 is designed to familiarize learners with the capability of Python's packages to do exactly this. It uses the calculation and analysis of a popular metric for crop growth measurement to demonstrate the usage of Python's *requests, Pandas* and *matplotlib* packages in completing agricultural data analysis problems. This workbook scrapes public weather data, preprocesses it for wrangling, calculates new parameters, and creates appropriate visualizations, thus automating data analysis for the end user.

Ag Data Analysis II: Problem Statement

Weather data is an important requirement in most if not all agricultural data analysis problems such as the design of field trials, farm management decision tools, crop growth prediction etc. The infrastructure for monitoring the weather for

agriculture includes government and third-party weather stations across the nation. The weather stations in the United States are standardized to a significant extent, especially stations managed by government agencies in the national and regional networks. Third-party or individual weather systems are designed to generate standard data outputs and work along with cloud data service providers to enable data storage and access. Automating the download, preprocessing, and wrangling of these public resources can create novel opportunities for end users interested in weather forecasts, crop growth predictions, and farming decisions among many other applications. Several applications which use public weather data for agricultural use already exist such as AgroClimateTools (High Plains Regional Climate Center, 2020) and Ag-Climate Tool (Corn GDD, 2017). Therefore, automating weather data analysis for agricultural problems is the need of the hour. To this end, the discussions in Demonstration Workbook 3 work towards exploring and analyzing the role of historical weather data in calculating and analyzing the trends in the development of corn.

Data Scraping

We have learned the basic data input-output methods in section 3.2.2, however, these methods have a high probability of error as they depend on manually downloading data for analysis. Downloading data manually gets cumbersome in the case of (i) large datasets from websites or (ii) periodical data downloads and this leads to errors during the wrangling of data for analysis. For example, (a) periodic data download is required in creating a daily or hourly weather data visualization for farms, or (b) data from a farm vehicle's controlled area network (CAN bus) can be used in analyzing the performance and planning logistics of farm activities in real-time. A human cannot be employed to download and update these data in real-time without expecting large errors. Hence, data acquisition jobs are frequently automated in data analysis problems. Data scraping is commonly used to download data directly from an online database via the internet. These data scraping methods can be simply initiated with a program execution in Python or be automated in time to make a completely automated system. Python has several packages that facilitate easy data scraping methods. *Requests* is one such package which allows scraping data in the hypertext transfer protocol (HTTP). Discussion 1 in Demonstration Workbook 3 illustrates the use of *requests* methods to scrape weather data from ACIS maintained by National Oceanic and Atmospheric Administration (NOAA) regional climate centers. ACIS hosts hourly updates on weather data and facilitates public sharing of historical data via APIs. APIs are interfaces used to communicate between two software applications. Data transfer APIs work between the applications shown in Figure 8 and set protocols for their communication. It standardizes the syntax for data retrieval from and data publishing to the host database. Each online database can create their unique APIs to serve their database.

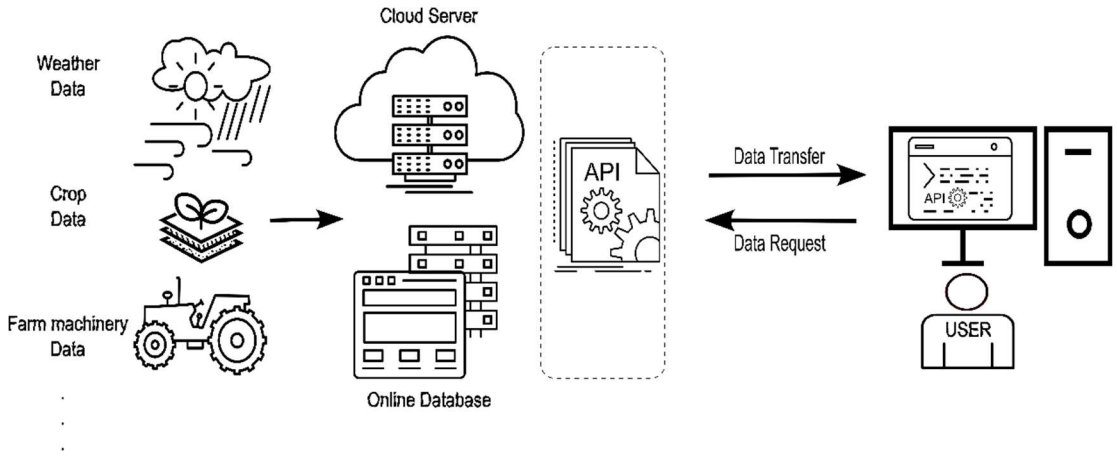


Figure 8. An illustration of API's role in scraping data from an online database or a cloud server.

API information is well documented by online databases, and this is the case with ACIS (ACIS, 2017). Figure 9 shows (a) the query used in Discussion 1 of the Demonstration Workbook 3 and (b) the query parameters generated in the box "Parameters (JSON)" in the ACIS query builder. The generated query can be copied into the Python code and used to request data. Users should try the query builder to generate different weather data queries, that can be copied to the Python code for practice.

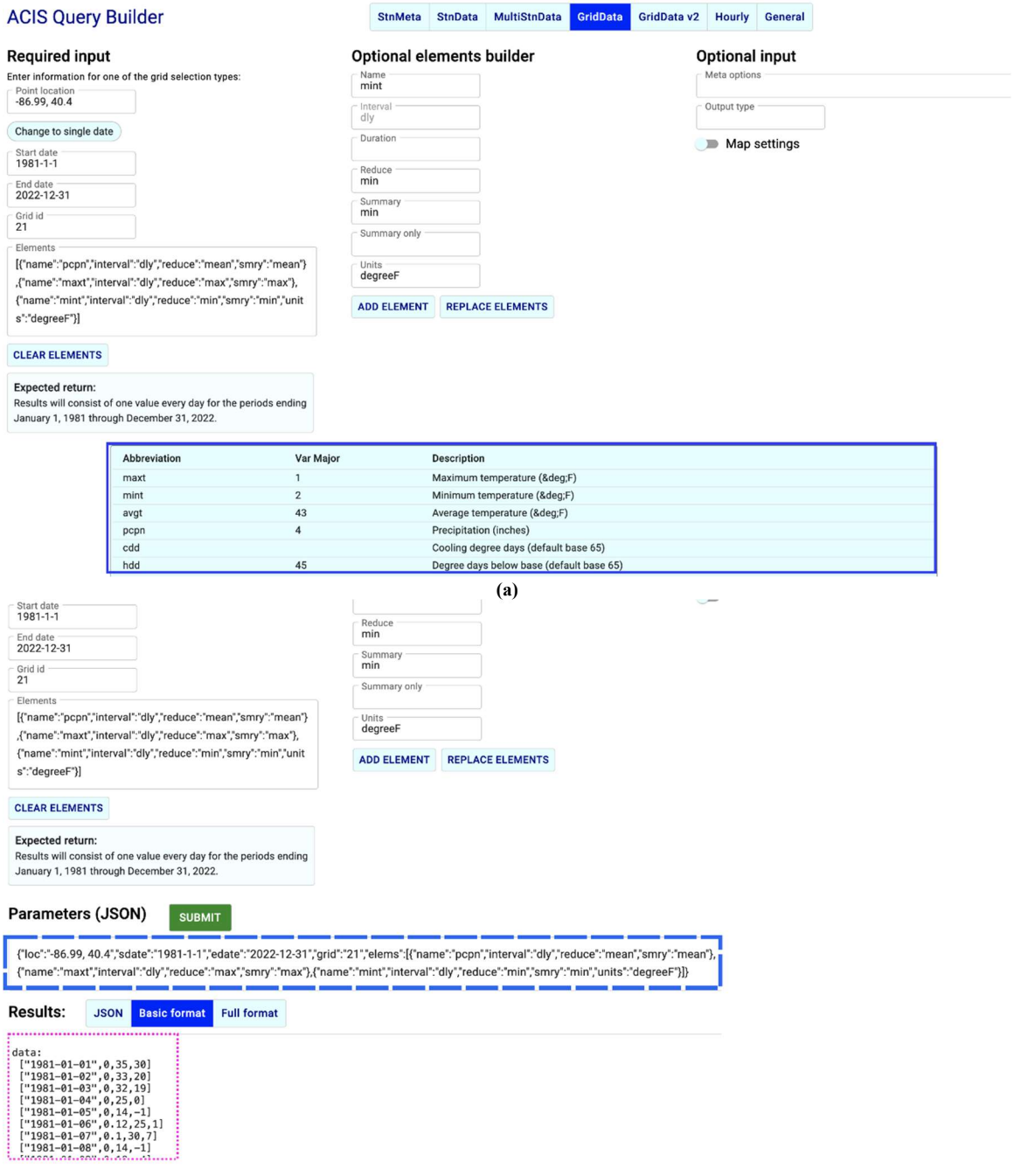


Figure 9. Illustrates a query in the ACIS query builder. Variables are selected using the table, marked in blue in subfigure (a). The generated query and example of the raw data format are also shown. These are marked in a dashed blue box and a pink dotted box respectively in the subfigure (b).

Data scraping requests to the ACIS web server require the location, date range, grid identification, and weather elements. The ACIS provides gridded weather data developed from the "*parameter-elevation relationships on the independent slopes model (PRISM)*" (PRISM Climate Group, 2014). The Demonstration Workbook 3 uses the geographical coordinates of the Purdue University Agronomy Research and Education (ACRE) farm to request ACIS-PRISM data. Daily temperature and precipitation data from January 1, 1981, till December 31, 2022, is scraped from the ACIS-PRISM database. The ACIS-PRISM web server responds to our request with data columns as JavaScript object notation (JSON) objects (JSON.org, 2013). The JSON file is a dictionary-type text data format which can store any data type. It uses the key-value pair format to store data. Each key is indexed in a JSON file. The keys are objects that can be used to extract the value paired with it. The downloaded raw weather data is parsed and converted into a CSV format in Discussion 1 of Demonstration Workbook 3. It should be noted that the JSON data is being converted to CSV for familiarity in this discussion, it is not required in wrangling data. The data scraped from the ACIS online database is now stored in the JSON object. A screenshot of the code

written to scrape the data from the ACIS database in Discussion 1, is shown in Figure 10. It can be saved into a file. It can also be directly used in the next steps of data wrangling. As the scraped data is 2-dimensional it can be stored in a 2D array, however, Python's *Pandas* package provides a better data type for 2D data. The *Pandas* package is discussed in the next sub-section.

```
# GPS coordinates of Purdue ACRE farm
lat = 40.4742259
lon = -86.9975974
# PRISM goes back to 1981
sdate = "1981-01-01" #start date for historical data
edate = "2022-12-31" #end date for historical data
# Make ACIS API request
w = requests.post('http://data.rcc-acis.org/GridData', json={
    "loc": f"{lon}, {lat}",
    "sdate": sdate,
    "edate": edate,
    "grid": "21",   # "21" is PRISM
    "elems": [
    # "maxt" is maximum temperature at interval daily
    #in degree Faherenheit and precipitation in inch
        { "name":"maxt", "interval":"dly", "units":"degreeF" },
        { "name":"mint", "interval":"dly", "units":"degreeF" },
        { "name":"pcpn", "interval":"dly", "units":"inch" }
    ]
})
# Parse the JSON response to a Python datatypes
w = w.json()
#use for loop to print the first 5 items in the retrieved weather data
for item in w['data'][0:5]:
    print("the retrived weather data: ",item)
```

✓ 25.2s MagicPython

```
the retrived weather data:  ['1981-01-01', 34, 30, 0.0]
the retrived weather data:  ['1981-01-02', 33, 18, 0.0]
the retrived weather data:  ['1981-01-03', 32, 18, 0.0]
the retrived weather data:  ['1981-01-04', 24, -2, 0.0]
the retrived weather data:  ['1981-01-05', 13, -3, 0.0]
```

Output

Figure 10. Illustration of an API request from a Python program. The output is displayed below the code cell. It shows each line of the scraped data is the selected weather elements for each day separated by a comma.

Pandas: Data Analysis Library

The data analytics package *Pandas (Pandas development team, 2020)* is introduced at this point in the course. *Pandas* has a very useful library of routines and methods for data-wrangling tasks. *Pandas* use "*DataFrame"* as their data object. DataFrame has a 2D tabular structure with heterogeneous data types. DataFrame is built upon NumPy's array objects and expands its capabilities to use heterogenous data and mutable size which makes it efficient in handling a large variety of datasets. The other unique data structure in *Pandas* is the series which is 1D tabular data with homogenous data types and mutable size. The two-dimensional tabular structure of the DataFrame resembles the tabular structure of data in Excel sheets which enables simple storing and effective visualizing of numerical and text data. *Pandas* can read and convert all datatypes like comma-separated values (CSV), JavaScript object notation (JSON), txt, shapefiles, and image files into *DataFrame*.

Discussion 2 in the Demonstration Workbook 3, illustrates a simple method to convert the JSON object from Discussion 1 in this workbook into a Pandas DataFrame. A screenshot of weather data stored in a Pandas *DataFrame* taken from Discussion 2 is shown in Figure 11. The column labels maximum temperature (maxt), minimum temperature (mint), and precipitation (pcpn) are kept the same as in the raw data downloaded from the weather database. These column labels are unique IDs and can be changed to other unique IDs at any time in the program. The "index" column in the DataFrame in Figure 11 is named and generated by default. Python creates an "index" column with all unique values when any DataFrame is created. It also creates unique column IDs when no column label is provided during DataFrame creation. DataFrame uses the "index" unique row values and column IDs to access its cell ID or value. It is interesting to note here that this DataFrame is also a "tidy" data format (Wickham, 2014).

index	date	maxt	mint	pcpn
0	1981-01-01	34	30	0
1	1981-01-02	33	18	0
2	1981-01-03	32	18	0
3	1981-01-04	24	-2	0
4	1981-01-05	13	-3	0
5	1981-01-06	25	-1	0.12
6	1981-01-07	30	5	0.1
7	1981-01-08	13	-1	0
8	1981-01-09	18	-4	0

Cell

Column labels

Row labels

Figure 11. An illustration of the tidy ACIS weather data in Pandas DataFrame object.

The ACIS weather data, now converted into the Pandas DataFrame (named "w" in the workbook) is ready for exploration and wrangling. Discussion 3 of the Demonstration workbook 3 illustrates the *head()* method which displays the top 5 rows of the *DataFrame*, and the *tail()* displays the last 5 rows of the DataFrame. The "*describe()*" method is used to tabulate the central tendency of the DataFrame in a single command. The implementation of *head()* and *describe()* is shown in Figure 12 (a) and (b).

```
w.head()
```

✓ 0.0s MagicPython

date	maxt	mint	pcpn
1981-01-01	34	30	0.0
1981-01-02	33	18	0.0
1981-01-03	32	18	0.0
1981-01-04	24	-2	0.0
1981-01-05	13	-3	0.0

Output

(a)

```
w.describe()
```

✓ 0.0s MagicPython

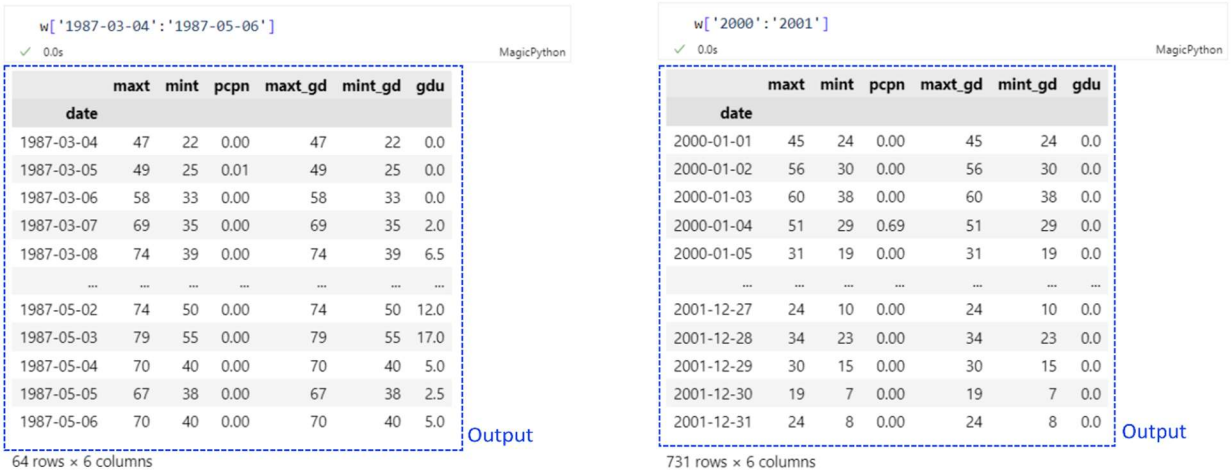
	maxt	mint	pcpn
count	15340.000000	15340.000000	15340.000000
mean	61.340743	41.285463	0.107096
std	21.012896	18.958502	0.289039
min	-11.000000	-24.000000	0.000000
25%	44.000000	28.000000	0.000000
50%	64.000000	42.000000	0.000000
75%	80.000000	57.000000	0.050000
max	103.000000	79.000000	4.490000

Output

(b)

Figure 12. Results from the implementation of (a) *head()* and (b) *describe()* on ACIS weather DataFrame.

Discussion 3 also works out an example to create a new column for growing degree days (GDD) value for each day using the average of columns "maxt" and "mint". Examples in Discussions 3 and 4 in the Demonstration Workbook 3 further describe the versatile range of operations enabled by the DataFrame object based on its indexing scheme and mutable character. Examples show efficient data retrieval from the ASIC weather DataFrame through label or position indexing. The ACIS weather DataFrame can be reduced in size by selecting, slicing, or grouping smaller sets of cells by values or indices. The pandastutor.com (Lau et al., 2023; Lau & Guo, 2022) is a valuable tool to visualize the *pandas* methods for better understanding. Pandas DataFrame is also equipped to apply functions to its subsets and is adept at performing a variety of time-series manipulations. Time series data such as weather data are convenient to manipulate with a datetime index, and *Pandas* "*to_datetime*" method is the simplest way to convert the date values from string to independent date, month, and year objects. These datetime objects can be used as the index to select values within a time range. Figure 13(a) and (b) show examples of selecting rows in a date range and a range of years taken from Discussion 3 of the Demonstration Workbook 3.



date	maxt	mint	pcpn	maxt_gd	mint_gd	gdu
1987-03-04	47	22	0.00	47	22	0.0
1987-03-05	49	25	0.01	49	25	0.0
1987-03-06	58	33	0.00	58	33	0.0
1987-03-07	69	35	0.00	69	35	2.0
1987-03-08	74	39	0.00	74	39	6.5
...	...	...	...	...	...	...
1987-05-02	74	50	0.00	74	50	12.0
1987-05-03	79	55	0.00	79	55	17.0
1987-05-04	70	40	0.00	70	40	5.0
1987-05-05	67	38	0.00	67	38	2.5
1987-05-06	70	40	0.00	70	40	5.0

(a)

date	maxt	mint	pcpn	maxt_gd	mint_gd	gdu
2000-01-01	45	24	0.00	45	24	0.0
2000-01-02	56	30	0.00	56	30	0.0
2000-01-03	60	38	0.00	60	38	0.0
2000-01-04	51	29	0.69	51	29	0.0
2000-01-05	31	19	0.00	31	19	0.0
...	...	...	...	...	...	...
2001-12-27	24	10	0.00	24	10	0.0
2001-12-28	34	23	0.00	34	23	0.0
2001-12-29	30	15	0.00	30	15	0.0
2001-12-30	19	7	0.00	19	7	0.0
2001-12-31	24	8	0.00	24	8	0.0

(b)

Figure 13. The output of selecting partial weather data by using the datetime index. The output of (a) selecting the rows from the *DataFrame* between two dates in 1987 and (b) selecting the rows from 2000 and 2001.

Additionally, *DataFrame* allows convenient data combination techniques through merging, appending, and joining, some of which are illustrated using the weather data in the following cells of Discussion 4 of the Demonstration Workbook 3. Discussions 3 and 4 implement matplotlib plots to illustrate the results of grouping, slicing, and merging methods of Pandas DataFrame object. Figure 14 is a screenshot of an example plot from Discussion 4 in Demonstration Workbook 3. It shows an error plot of the average of daily maximum temperatures from 1981 to 2022. The required data for this plot was created by grouping the ACIS weather DataFrame by day using the "*group by"* method and selecting the maximum daily temperatures over time. The maximum daily temperature and its average from 1981 to 2022 are used to draw the error bar plot. Figure 14 also has a graph of the average daily growing degree days/units (GDU) over the years from 1981 to 2022. The data for this graph was created using the Pandas "*group by"* method to group the GDU by day and then calculate the mean of daily GDU from 1981 to 2022.

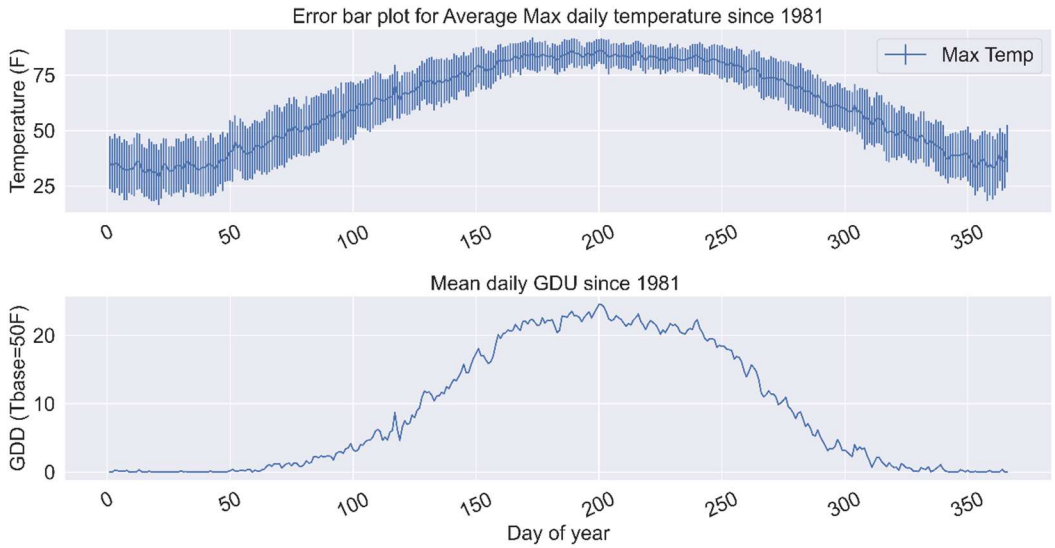


Figure 14. Visualization of results from the application of Pandas data wrangling and analysis methods on ACIS weather DataFrame.

The final resulting ACIS weather DataFrame with additional columns and standardized data formatting can be saved on the local machine for reference. *Pandas* can save data on disk in any required format using a single line statement shown in Discussion 4 of the Demonstration Workbook 3.

Ag Data Analysis II: Estimating Growth Stages of Corn

The resulting ACIS weather DataFrame from Discussion 4 has undergone the required preprocessing and wrangling for calculating the growth stages using GDU for crops. GDU calculations for determining the crop development stages is an important crop growth analysis technique used to create appropriate crop management decisions (Nield, Ralph E. and Newman, James E., 1990). GDU is calculated from daily air temperatures by using the relationship, $\text{GDU} = \frac{Tmax + Tmin}{2} -$

$Tbase$, where $Tbase$ =50F is the lower threshold for air temperature (T). Air temperatures above 50F contribute towards the GDU of corn and the accumulation of GDU over time is used to predict the vegetative and reproductive growth stages of corn (M.R.C.C. Purdue, 2013).

Discussion 5 of the Demonstration Workbook 3 illustrates the use of GDU in calculating and analyzing the growth stages of corn. Functions to calculate the vegetative index (V) and reproductive index (R) of corn are written. The calculated values of GDU, V and R are stored in new columns in the ACIS weather DataFrame. The GDU, V and R columns are used to create multi-plot figure in matplotlib, shown in Figure 15. These values can also be used to calculate the time to silking and black layer for corn.

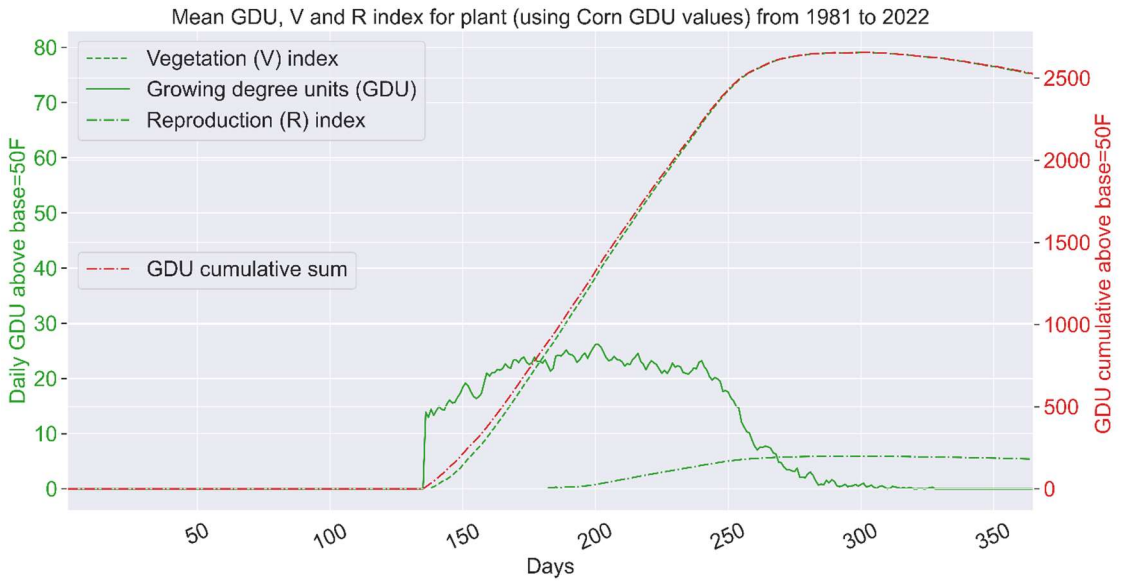


Figure 15. This plot displays the daily GDU, cumulative GDU, markers for growth stages, vegetative index, and reproductive index using the average temperatures of each day measured at Purdue ACRE farm in West Lafayette, IN from 1981 to 2022.

At the end of the Demonstration Workbook 3 learners should be able to program the essential steps of data acquisition, wrangling, analysis, and display results in clear and informative visuals using requests, Pandas and matplotlib methods for publicly available agricultural data.

DEMONSTRATION WORKBOOK 4: CHUTES AND LADDERS

The last Demonstration Workbook of this course module illustrates another important aspect of programming — mathematical modelling, made easy by Python's packages. The discussions in this workbook use simple mathematical models implemented in games of chance. The Chutes and Ladders is a children's game where the ladders are used as a reward to move forward, and chutes are penalties that take the players back. It is a simple game to simulate as there are no player choices to consider. An example Chutes and Ladders board, used in the Discussions of Demonstration Workbook 4 is shown in Figure 16.

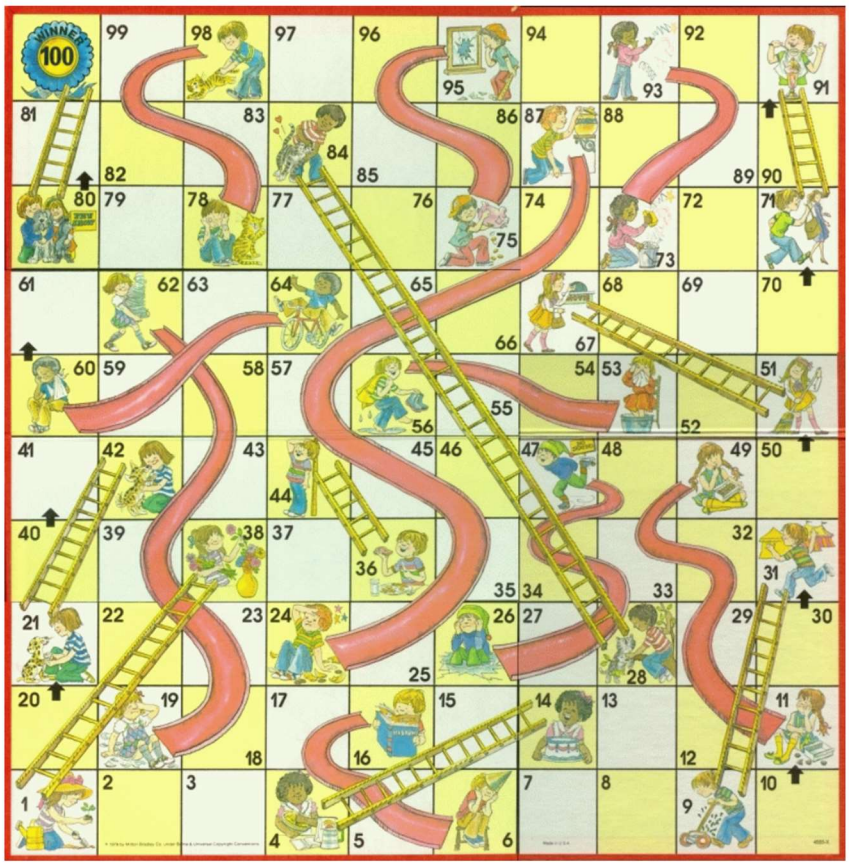


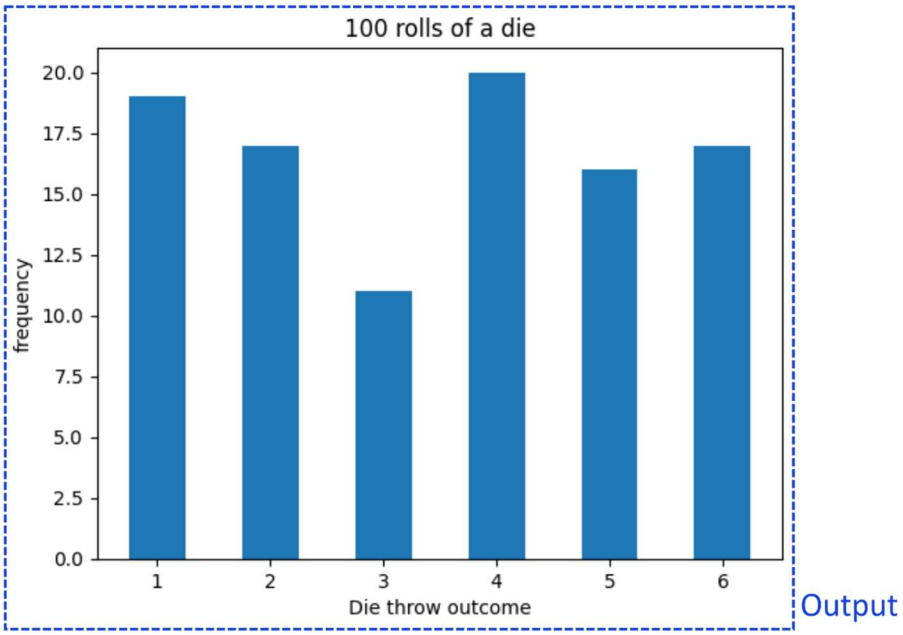
Figure 16. A sample Chutes and Ladder board game.

The rules of the game are described in Discussion 1 of the Demonstration Workbook 4. Discussion 2 explains the random number generator of *NumPy*. It is used to generate random numbers in the declared data type and a given range. NumPy uses pseudorandom number generator algorithms to generate random numbers from probability distributions in the given range of values. The *seed* variable is used to select an initial value from which the random numbers will be generated and to generate the same random numbers in each run if required. Discussion 3 of the Demonstration workbook 4 illustrates the use of dictionary variables to represent the board design of the game. In this dictionary, the "*key: value"* pair stores the "*starting number: end number"* pair of ladders and chutes from a sample game board shown in Figure 16. Discussion 3 also illustrates the function written to model random throws of a die and compute the new position of the player after each turn. The die-throw function prints random integers in the range of 1-6. This simulates the results of random dice throws in the game. The histogram of each dice outcome in 100 throws is plotted. The histogram analysis is key to testing the probability of completely randomizing dice outcomes. In a completely randomized scenario, the frequency bars of all the values of dice should have equal height. The code to generate 100 dice throws and the histogram of the frequency of each dice output are shown in Figure 17 (a) and (b) respectively. These analyses are useful to ascertain true randomness during simulation.

```
# NumPy's random number generator is used to generate random numbers in Python.
np.random.seed(5)# seed is used to initialize random number generator
rolls = np.random.randint(1,7,100) # 100 random numbers between 1 and 6
plt.hist(rolls, bins=range(1,8,1),align='left',rwidth=0.5) # histogram of the rolls
plt.title('100 rolls of a die')
plt.xlabel('Die throw outcome')
plt.ylabel('frequency')
```

✓ 0.1s MagicPython

(a)



(b)

Figure 17. The subfigure (a) shows the code snippet to roll a die twice and display its output in the terminal, and (b) shows the output histogram of 100 rolls with dice face values on the X-axis and the frequency on the Y-axis.

Discussion 4 illustrates the function written to model a 2 player Ladder and Chutes game until a player wins. It references the game board dictionary variable, the function to model the die roll and compute player position from discussion 3 and creates a program to model the entire game. This program also plots the result of a game using matplotlib shown in Figure 18. The plot shows that the game ended in 33 turns with Player 1 winning the game. The analysis also shows that Player 1 dominated the game throughout even if they started slower than Player 2.

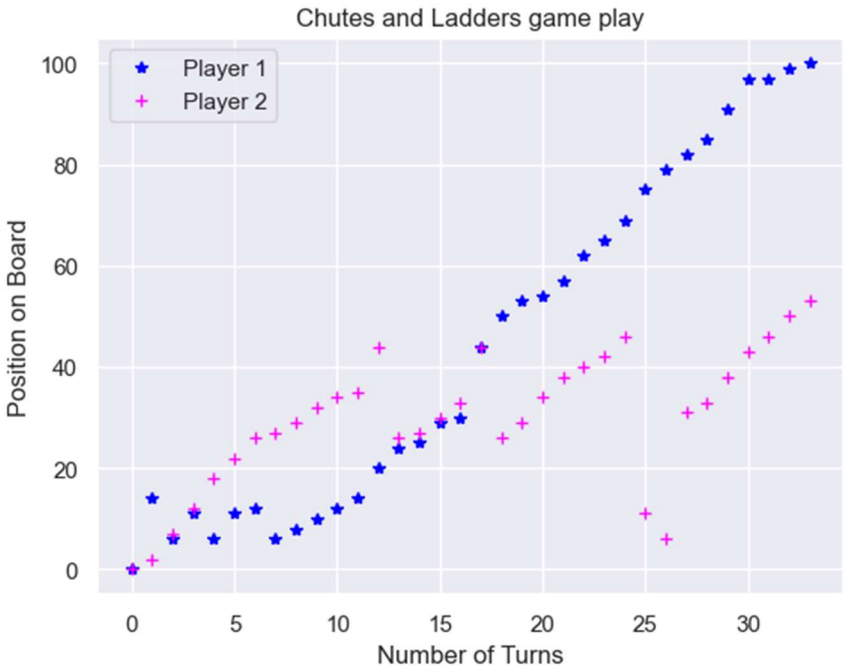


Figure 18. Illustrates the results of a two-player chutes and ladder game simulated and visualized in Python. The X-axis denotes the turn, and the Y-axis shows the position of both players after each turn.

Discussion 4 also calculates the average number of turns used in the 2-player Chutes and Ladder game when played 1000 times. The code written to simulate 1000 runs of the game also calculates the average number of turns used to win the game shown in Figure 19 (a). It also plots a histogram of the frequency of probable number of turns used to win in 1000 games,

as illustrated in Figure 19 (b). More analysis examples are demonstrated in Discussion 4 of the Demonstration Workbook 4 to show the effects of board design on games using multiple-game analysis.

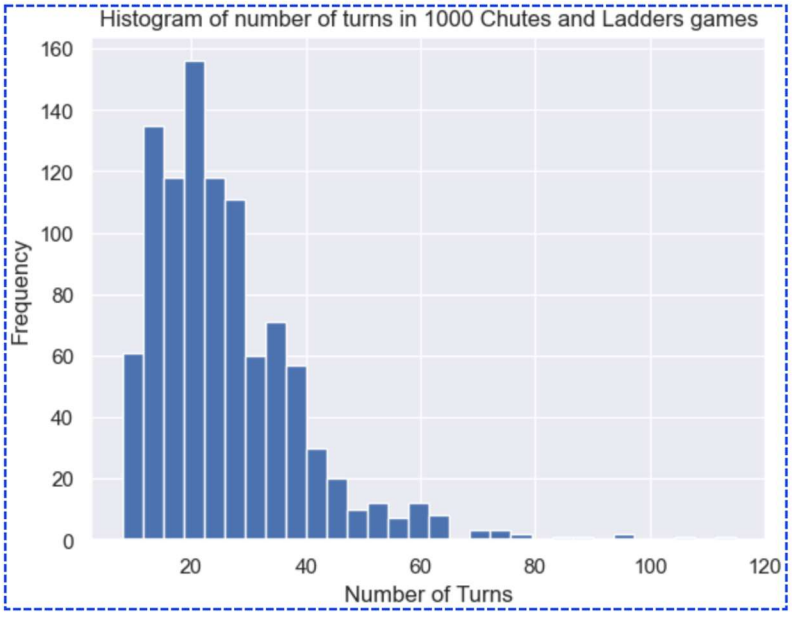
```
# Convert game code to function

def CandL_play_a_game(CandLTable): ···
Ngames = 1000 # Number of games to play
Np1wins = 0# Initialize the number of wins for each player
Np2wins = 0# Initialize the number of wins for each player
GameLengths = np.array([])
for k in range(Ngames): ···
print('Average number of rounds per game is {:.2f}'.format(np.mean(GameLengths)))
print('Player 1 won {} of {} games'.format(Np1wins,Ngames))
print('Player 2 won {} of {} games'.format(Np2wins,Ngames))
plt.hist(GameLengths, bins=30)
plt.xlabel('Number of Turns')
plt.ylabel('Frequency')
plt.title('Histogram of number of turns in 1000 Chutes and Ladders games')
```

✓ 0.6s MagicPython

```
Average number of rounds per game is 26.02
Player 1 won 491 of 1000 games
Player 2 won 509 of 1000 games
```

Output

(a)



Output

(b)

Figure 19. Subfigure (a) shows the code snippet written to simulate 1000 2-player Chutes and Ladder game and print the summary results as output in the terminal. The subfigure (b) shows the output histogram of the number of turns used to win, in 1000 simulations of the 2-player game.

At the end of the Demonstration Workbook 4, learners should be able to employ Python's packages to simulate simple mathematical models and analyze their performances using visual and statistical metrics. For example, the result analysis of the chutes and ladder game could be repeated with a different board design to compare its effects on the length of the game or the probability of winning. It would not be much of a stretch to alter the code to design another children's board game such as Hi Ho! Cherry-O for simulation practice.

PRACTICE WORKBOOK B: VISUALIZATION USING MATPLOTLIB.

Practice Workbook B is designed to guide learners in creating anatomically correct and clear figures using the *matplotlib* package. The data used in this workbook is the same as the data discussed in the Demonstration Workbooks 2 and 3. The

examples illustrate techniques to program visualizations for mathematical functions, plots with shared axes, and modify figure elements. The solutions for the Practice Workbooks are in the module repository.

On completion of the Demonstration Workbook 3, 4 and Practice Workbook B, learners can:

i. Make an educated decision on required packages and install them in their Python environment.
ii. Search public APIs to download, process, and write data on disk.
iii. Implement *Pandas* methods and functions for data handling and manipulation.
iv. Simulate random variables and analyze predictions/outcomes based on simple mathematical models.
v. Include dense information in clear visualization using the *matplotlib* package.

DISCUSSION

In this paper, we present a complete guide for engaging students with no prior programming experience in a concise Python learning course. The structure of the Jupyter Notebooks used in creating the workbooks allows students to practice writing codes while simultaneously studying the sections of this paper. To enhance clarity, the paper references discussions from the workbooks with the relevant topics. The GitHub repository associated with this paper is publicly available and will be actively maintained in the future. Additionally, a set of guidelines to build custom Python tutorials based on this course is described in the appendix. Instructors can follow these guidelines to create advanced Python tutorials for agricultural data analysis and other environmental data analysis problems.

REFERENCES

ACIS. (2017). Applied Climate Information System. Indiana, USA: Regional Climate Center. Retrieved from https://www.rcc-acis.org/docs_gridded.html

Corn GDD. (2017). Midwestern Regional Climate Center. *Purdue University*. Retrieved from https://mygeohub.org/groups/u2u/purdue_gdd

Guido van Rossum, P. d. (2018). The PythonTutorial. Retrieved from https://bugs.python.org/file47781/Tutorial_EDIT.pdf

Guo, P. (2014). Python Is Now the Most Popular Introductory Teaching Language at Top U.S. universities. *Communications ACM*. Retrieved from https://cacm.acm.org/blogs/blog-cacm/176450-python-is-now-the-most-popular-introductory-teaching-language-at-top-us-universities/fulltext

Guo, P. (2021). Ten million users and ten years later: Python tutor's design guidelines for building scalable and sustainable research software in academia. *The 34th Annual ACM Symposium on User Interface Software and Technology*, (pp. 1235–1251). Retrieved from https://pythontutor.com/

Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., . . . Oliphant, T. E. (2020). Array programming with NumPy. *Nature, 585*, 357–362. https://doi.org/10.1038/s41586-020-2649-2

High Plains Regional Climate Center. (2020). *ACIS Climate Maps*. Retrieved from https://hprcc.unl.edu/maps.php?map=ACISClimateMaps

Hunter, J. D. (2007). Matplotlib: A 2D graphics environment. *Computing in Science & Engineering, 9*, 90–95. https://doi.org/10.1109/MCSE.2007.55

JSON.org. (2013). *JSON (JavaScript Object Notation)*. Retrieved from JSON (JavaScript Object Notation): https://www.json.org/json-en.html

Jupyter, P., Bussonnier, M., Forde, J., Freeman, J., Granger, B., Head, T., . . . Willing, C. (2018). Binder 2.0 - Reproducible, interactive, sharable environments for science at scale. In F. Akici, D. Lippa, D. Niederhut, & M. Pacer (Ed.), *Proceedings of the 17th Python in Science Conference*, (pp. 113-120). https://doi.org/10.25080/Majora-4af1f417-011

Krogmeier, J. V., Jha, S., Balmos, A. D., & Buckmaster, D. R. (2024). PythonInDigitalAg. https://doi.org/10.5281/zenodo.10477638

Lau, S., & Guo, P. (2022). Pandas tutor visualizes how Python code transforms dataframes. Pandas Tutor - visualize Python pandas code. Retrieved from https://pandastutor.com/

Lau, S., Kross, S., Wu, E., & Guo, P. J. (2023). Teaching Data Science by Visualizing Data Table Transformations: Pandas Tutor for Python, Tidy Data Tutor for R, and SQL Tutor. *Proceedings of the 2nd International Workshop on Data Systems Education: Bridging education practice with education research*, (pp. 50–55).

M.R.C.C. Purdue. (2013). degreeday_description. Retrieved from https://mrcc.purdue.edu/CLIMATE/Station/Daily/degreeday_description.html

Nield, R. E. and Newman, J. E. (1990). Growing season characteristics and requirements in the Corn Belt. Purdue University Cooperative Extension Service. Retrieved from https://www.extension.purdue.edu/extmedia/nch/nch-40.html

Pandas development team. (2020). pandas-dev/pandas: Pandas. *pandas-dev/pandas: Pandas*. Zenodo. https://doi.org/10.5281/zenodo.3509134

PRISM Climate Group. (2014). *Oregon State University*. Retrieved from PRISM: https://www.prism.oregonstate.edu/

Python Module Index. (2023). Python Official Documentation. Retrieved from https://docs.python.org/3/py-modindex.html#cap-p

Python Packaging Authority. (2011). Warehouse. Retrieved from https://github.com/pypi/warehouse

Python Software Foundation. (2023). Built-in Functions. Retrieved from https://docs.python.org/3/library/functions.html

Python Software Foundation. (2023). The Python Tutorial. Retrieved from https://docs.python.org/3.12/tutorial/index.html

Python Software Foundation. (2023). Using the Python Interpreter. Retrieved from https://docs.python.org/3/tutorial/interpreter.html

PythonModuleIndex. (2023). Python Module Index. Retrieved from https://docs.python.org/3/py-modindex.html

Raphael, B. (1966). The structure of programming languages. 67–71. https://doi.org/10.1145/365170.365175

Siegfried, R. M., Herbert-Berger, K. G., Leune, K., & Siegfried, J. P. (2021). Trends Of Commonly Used Programming Languages in CS1 And CS2 Learning. *2021 16th International Conference on Computer Science & Education (ICCSE)*, (pp. 407-412). https://doi.org/10.1109/ICCSE51940.2021.9569444
Steponaitis, S. (2020). OrganizationsUsingPython. *OrganizationsUsingPython*. Retrieved from https://wiki.python.org/moin/OrganizationsUsingPython
The Data Mine. (2020). The Example Book. The Data Mine. Retrieved from https://the-examples-book.com/programming-languages/python/indentation
The pip developers. (2008). *Python Package Authority*. Retrieved from https://pip.pypa.io/en/stable/installation/
The Python Standard Library. (2023). math-Mathematical functions. Retrieved from https://docs.python.org/3.11/library/math.html#module-math
The Python tutorial. (2023). Data Structures. *Python Software Foundation*. Retrieved from https://docs.python.org/3/tutorial/controlflow.html
The Python tutorial. (2023). More Control Flow Tools. *Python Software Foundation*. Retrieved from https://docs.python.org/3/tutorial/controlflow.html
U.S. National Agricultural Statistics Service NASS. (1998). Quick Stats. Retrieved from https://quickstats.nass.usda.gov/
Wickham, H. (2014). Tidy Data. *Journal of Statistical Software, 59*, 1–23. https://doi.org/10.18637/jss.v059.i10

APPENDIX

SUGGESTIONS FOR INSTRUCTORS

This teaching module facilitates sequential learning of Python programming language and caters to the needs of learners and instructors alike. It provides a well-documented setup and installation file for Python software and introduces a simple programming environment for local machines while also offering an alternative option to progress through the module online without any setup. This feature was thoughtfully implemented to overcome resource disparity among learners. The Demonstration Workbooks in the module are intentionally aligned to follow the organic progression of learning programming in Python, tailored to beginners. Emphasis has been placed on the careful arrangement of discussions in each workbook to ensure a seamless transition from fundamental to advanced topics.

The Demonstration Workbooks are annotated with comprehensive comments, ensuring that they are self-explanatory and easy to understand. Additionally, important links are incorporated in the markdown section, serving as resource guides for instructors to access relevant background material. These resources can be used to expand upon the workbooks and customize them for specific needs. The Practice Workbooks are primarily intended for assessment and come complete with solutions conveniently housed in the module repository. This exemplifies an effective learning experience and empowers instructors to develop better workbooks by using ideas from the module workbooks.

Suggestions for instructors to develop and implement tutorials based on this module:

i. Focus on ease of accessing the environment used to compile, share, and store programs.
ii. Carefully plan the flow of the tutorials to transition from simple to complex language constructs, functions etc.
iii. Provide enough examples of a concept and provide one good reading material on it, generally referring to the official document is the best practice.
iv. Add explanatory comments specific to the programs or assign them as student exercises.
v. Use examples to encourage simultaneous coding practice for students. For example, omit the solution workbook from this course module before sharing it with students, and provide incentives in the form of hints for the problems in Practice Workbooks. These hints can be shared on completion of a programming task such as the successful run of examples from the Demonstration Workbooks.
vi. The two aspects envisioned in the course module are (a) learning Python, and (b) learning to analyze agricultural data in Python. The student's understanding of both aspects should be evaluated. Instructors can use the assessment tasks in doing the same. The question in assessment task 2.1 is shown in Fig. 20. In this task, the student must plot a histogram of the yield data used in Demonstration Workbook 2.

Assessment task 2.1: Navigate to the python graph gallery and plot the histogram for yield data and show the mean and median of the yields on a histogram plot. Write a markdown below your code to explain

- the number of bins selected
- the mean and
- the median of the yield data.

add median/mean line in the Yield in bushels per acre vs Year plot, to challenge yourself !!

The output plot could look similar to this.

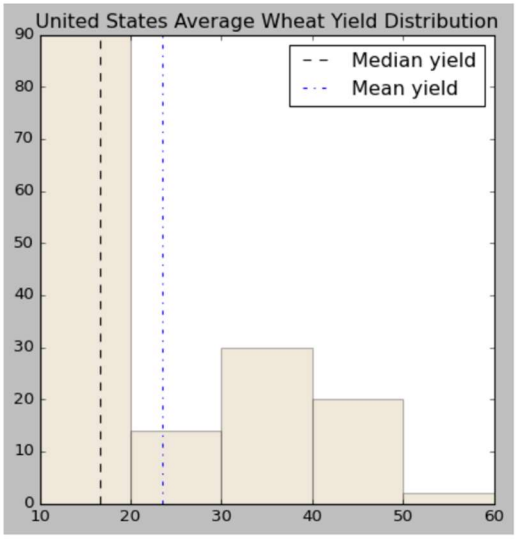


Figure 20. A sample assessment task from the Practice Workbook B.

The challenges in this task are (i) to create a well-labelled histogram in Python, (ii) choose a suitable bin size/range for the data, (iii) to write the rationale for the bin size/range in a markdown cell or comments, and (iv) to add advanced graph elements such as vertical lines, line style etc. in the figure. The rationale behind the bin size/range should be discussed by the instructor to evaluate the student's understanding of the yield. The instructor could also give a preview of the histogram plot from the exercise and evaluate the student's plot on each graph element added correctly.

vii. Use relevant datasets to demonstrate the capabilities of Python's language construct, functions etc.