



### **Cataloging Data**

Names: Simber, Chris, author.

Title: Python Programming: Basics to Advanced Concepts

Advanced Programming Workshop

Subjects: Python (Computer Program Language)

Chris Simber

Assistant Professor of Computer Science

Rowan College at Burlington County

Author contact: [csimber@RCBC.edu](mailto:csimber@RCBC.edu)



**Attribution-NoDerivs**

**CC BY-ND**

This work is licensed under CC BY-ND 4.0. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nd/4.0/>

# Introduction

This book is intended for use in a programming course in Python for students who are familiar with computer programming in another language such as C++ or Java. It follows the flow of a standard text for a programming language highlighting Python specifics and differences. This means that it moves quickly from variables to multi-file, multi-window advanced programming in a *how to implement things they know in Python* perspective. The goal is to provide students with the differences that can be expected when programming in Python.

The text is designed for instruction in a course in which students develop a semester-long project in the language, but can be used for courses that require multiple programs as well. The classroom format for a semester-long project is lecture followed by a collaborative or team workshop. Projects have six milestones for design and development with documentation submissions and presentations of running software. Additional project instructions are available from the author.

The examples within the chapters follow the PEP 8 Style Guide for Python Code and reinforce the material introduced while building on previous material covered. They provide the information necessary to meet each milestone in the project chronologically. Again, a comprehensive, semester-long project is in view. For this reason, there are no end-of-chapter reviews, summaries, or questions. However, the chapter exercises are numbered for clarity using a shaded box, and can be used for assignment purposes. Beginning in Chapter 5, the examples build the Data Analysis Project in Appendix C, however all projects have similar requirements. There are accompanying slides for instruction.

The Python version in use at the time of this writing is 3.9.0. The modules utilized include Tkinter, random, tkinterFileDialog, filedialog, webbrowser, numpy, pyplot, animation, PhotoImage, matplotlib, and winsound.

The Integrated Development Environment (IDE) selected is IDLE which accompanies Python when downloaded. IDLE's interface differs from most IDEs and provides students with a new experience in development and debugging that is not overtaxing yet instructional. IDLE has been selected intentionally for this purpose.

Instructions for obtaining and installing Python with IDLE are provided in Appendix A. Instructions for using the PIP installer which is included in Python are provided in Appendix B. Links to the Python web site, Python Tutorials, and the PEP 8 Style Guide are included in Appendix D.

#### Acknowledgements:

I would like to thank the following students for their technical and editorial review and suggestions. During a busy semester they found the time to provide valuable recommendations and editing contributions.

Christian Larcomb

Elizabeth Meyeroff

#### New in this edition:

Chapter 7 – added window centering, bringing windows to the front, and ending programs when windows are closed

Chapter 12 – an option list example 12.2A was added that uses “trace”

Chapter 13 - the tkcalendar module was added

Chapter 14 – the new Python format specifier was added

# Contents

Chapter 1	Python Programming & Process	1
Chapter 2	Python Language Specifics	9
Chapter 3	Getting Started in Python	17
Chapter 4	Decisions, Logic, Loops, and Functions	23
Chapter 5	GUI Design & Development	35
Chapter 6	File Handling	49
Chapter 7	Multiple Windows & Design	59
Chapter 8	Data File Design & Data Handling	69
Chapter 9	Strings, Lists, and Tuples	73
Chapter 10	Remove, Modify, or Hide Controls	79
Chapter 11	Main Interface GUI	83
Chapter 12	Menus and Button Groups	87
Chapter 13	Date and Time	97
Chapter 14	Displaying Data	101
Chapter 15	Python Modules	115
Chapter 16	File Dialogs, HTML, and Animation	121
Appendix A	Installing Python with IDLE	131
Appendix B	the PIP Installer	133
Appendix C	Data Analysis Project	135
Appendix D	Resource Links	141
Index		

“Five minutes of design time, will save hours of programming” –  
*Chris Simber*

# Chapter 1

## Python Programming & Process

Python is an interpreted, high-level, general-purpose language. It was created by Guido van Rossum and introduced in 1991, and emphasizes code readability and is similar in many respects to pseudo-code. The name Python comes from the famous British comedy Monty Python's Flying Circus.

Most programmers are familiar with *compilers* which translate high-level languages into machine language. Python uses an interpreter.

An *interpreter* translates and executes instructions in a high-level language. It interprets one instruction at a time and executes it. There is no separate machine language program.

Note: There are applications such as *PyInstaller* that package Python programs into stand-alone executables.

From a programmer standpoint, the process of interpret-execute removes the compilation step of some other languages that performs error and type checking. Integrated Development Environments for Python have evolved to help in this regard, and provide comparable support.

Interpreted languages tend to be slower than direct native machine code, and can be reverse engineered more easily, so their use should be restricted to areas where this is not an issue.

## A second Language

It is recommended that programmers be proficient in one language, familiar with others, and not be intimidated by any. Programming trends warrant a broader knowledge in computing than a single language or development environment provides.

### Programming Trends

- Server side (cloud-like) 1980s
- Stand-alone executables 1980s thru present
  - dramatic increase in available software
  - evolution of interfaces
- Web applications 1990s thru present
- Cloud applications (server side) 1993 thru present
- All of the above Today

The extensive use of the Graphical User Interface (GUI) and network and internet utilization including internet interfaces and transactions increase the need for multi-language programmers. Maintaining existing programs in various languages is a major area of the computer programming industry as well. For example, FORTRAN has been used in math and science, COBOL for business and finance, C and C++ in many areas, Java in web applications, and so on. At the time of this writing there were approximately 250 programming languages in use. Some of these have been used extensively, others not so much. Each has benefits and limitations as well as a following, proponents, and detractors.

Given what you know about programming: variables, functions/methods, classes and objects, logic, flow of control, algorithm development, etc., adapting to Python should not be difficult.

As an example, a simple output statement in C++, Java, and Python. Note the similarities.

<b>C++</b>	<b>cout &lt;&lt; "This program computes ..." &lt;&lt; endl;</b>
<b>Java</b>	<b>System.out.println("This program computes...");</b>
<b>Python</b>	<b>print('This program computes ...\n')</b>



Here are a few examples of Python code with comments for explanation.

```
# requesting and obtaining input, and storing it in a float variable
tempF = float(input('Enter the temperature in Fahrenheit: '))

# computation in Python raising windSpeed to the 0.16 power using ""
wind_chill = 35.74 + (0.6215 * tempF) - (35.75 * (wind_speed**0.16)) + \
    0.4275 * tempF * (wind_speed**0.16)

# displaying formatted output in Python
print('The wind chill factor is ', + format(str(wind_chill, '.2f')))
```

There are some key differences in the Python lines above from other languages. Variables are not declared by type, comments begin with the pound sign “#” (octothorp), and the use of the backslash at the end of the line computing *wind\_chill* is a line break in the code that does not affect execution. These and other differences will be described in detail and highlighted throughout the text.

## The Agile Development Process and Design and Development

As with any language, increasing design time shortens development, testing, and debugging time. The design tools typically utilized in other languages can be used with Python as well: design documentation, pseudo-code, Story Boards, IPO (Input Processing Output) documents, flowcharts, Unified Modeling Language (UML) diagrams, etc. It is often said that “Five minutes of design time can save hours of development and debugging.” This is a true statement.

Development cycle and IDE tools can be used as well including: text editors, watch windows, error alerts, breakpoints, comments, and output statements.

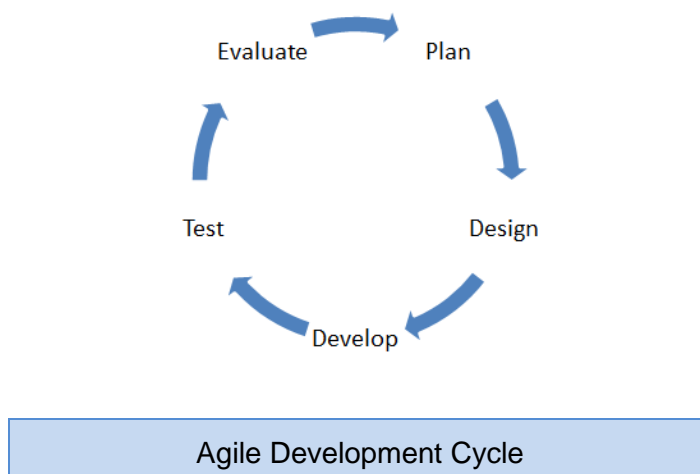
Tools for software teams and software project managers are commonly used in industry to plan and measure project progress, and to provide visibility into the design, schedule status, cost, and quality of the code. The **Agile Development Process** is a popular method in use today. Agile processes go by various names, but all are iterative and incremental software methodologies that lend themselves to Python development.

The most popular agile methodologies include:

- Scrum – regular meetings, periodic cycles called sprints
- Crystal - methodology, techniques, and policies
- Dynamic Systems Development Method (DSDM)
- Extreme Programming (XP)
- Lean Development
- Feature-Driven Development (FDD)

The **Scrum** methodology is further explained in terms of **sprints** to align with the milestone tempo of this text for project design and development.

A key component of the Agile Development Process is a sprint. Sprint meetings occur periodically (usually weekly or twice monthly) and include a review and planning event. Tasks completed from the previous sprint plan are reviewed, and completed work is demonstrated to stakeholders for feedback and approval. The tasks that were not completed from the previous sprint plan is reviewed with a course of action (re-plan). The scope of work that will be completed during the next sprint cycle is planned, and engineers are assigned to the tasks.



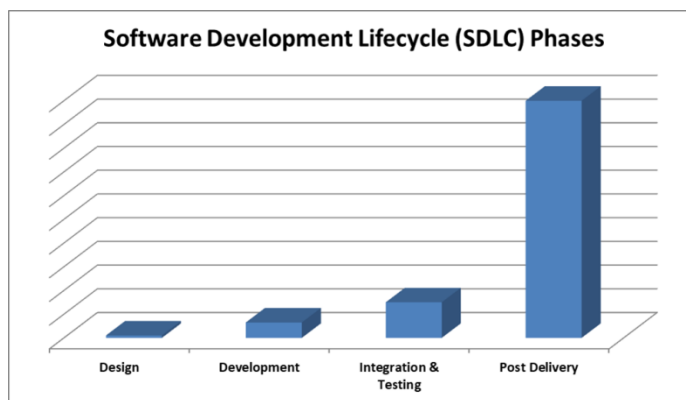
## Requirements

Prior to the planning and design phase, a complete understanding of what the program is supposed to do is needed. How it will do what it is supposed to do will be determined as the design phase is completed during the software development phase. *Requirements decomposition* is the act of discerning in

detail from the requirements what the program is to accomplish. This process also assists in decomposing the project into manageable “chunks” in terms of schedule and team assignment for development.

## Design

As the requirements are decomposed and documented, the design phase begins, and the break-down of required tasks and logical steps in the program are developed. Design is a very important part of the software development cycle because of the cost escalation of changes and bug fixes further on in the process. This is highlighted in the chart below from the IBM Systems Sciences Institute.



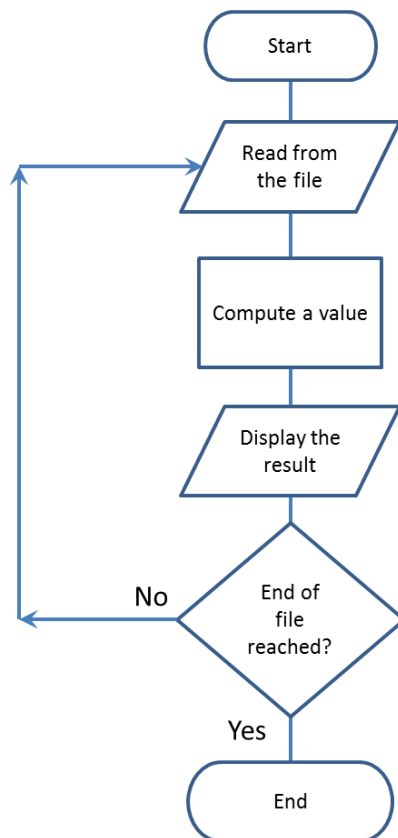
Cost Increase of Fixing Errors by Phase

Software engineering tools that assist in this process include pseudo-code, and flowcharts that graphically show the order of operations. Consider a program to read data from a file, compute a value, and display the results. The pseudo-code for the solution might be:

- Step 1. Start the program
- Step 2. Read data from the file
- Step 3. Compute the value
- Step 4. Display the output
- Step 5. End of file reached?
  - If No, go back to Step 2
  - If Yes, go to Step 6
- Step 6. End the program

The pseudo-code steps above do not include opening and closing the file. They might be considered obvious steps in the process. The level of detail is subjective. Rather than use a document, pseudo-code for portions of the program could be typed into the text editor of the IDE as comments. Later, they can either be replaced with actual code or kept in some form as a comment to the code.

Since we think in pictures and not text, a flowchart provides a faster and clearer depiction of the algorithm and logic. If we are simply ensuring that we have a robust algorithm and haven't missed any steps, then flowcharts can be sketched quickly on a piece of paper and discarded after the program is testing correctly. If the flowchart will be used later, or is part of the deliverable product, then a flowcharting application such as LucidChart<sup>®</sup> would be used. It is common for larger organizations to divide the design and development tasks into teams or to subcontract the software development out-of-house. In these instances, flowcharts are often required to be delivered to the development team or subcontractor together with specific requirements for the code. A flowchart for the example algorithm is shown below.



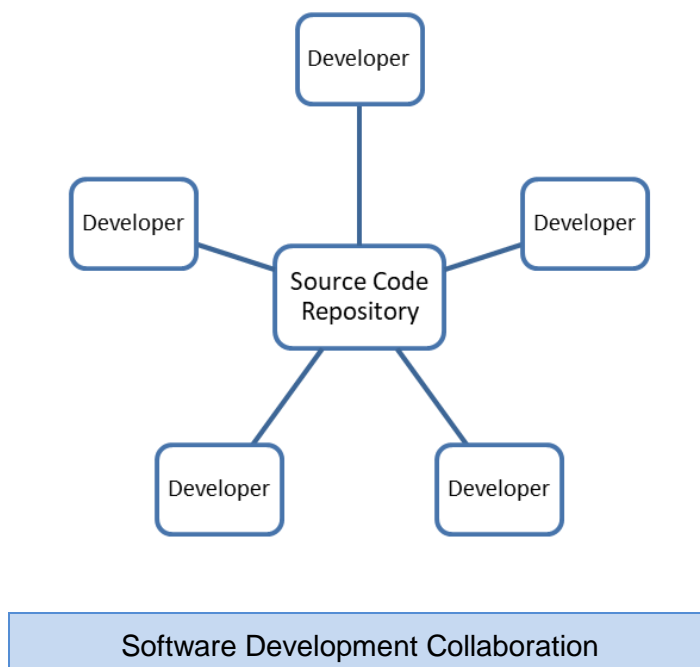
File Reading Flowchart

Flowcharts can ensure that steps in the process haven't been overlooked and that there is a complete understanding of the operational flow of the program.

Many software engineers use a combination of these tools. Pseudo-code may be used for a high-level description of the program or a program area, and a flowchart might be used for more complex sections. Either way, the goal is to have a comprehensive understanding of the requirements at every level to ensure that the final product meets the requirements.

## Development

Once a design is complete (or nearly complete since some aspects of the solution may not be knowable during design), the development phase begins. Often the development of a program is divided among multiple programmers and requires collaboration and regular discussion to ensure a cohesive solution. To enable multiple people to work on the same program at the same time, a *Configuration Management System (CMS)* is used with a source code repository that stores all of the programs files.



Programmers access this repository to obtain a copy of a file and add functionality or make modifications to the code. The code is written in the copied file, and this changed file is tested with the other files in the source code

repository. After testing, the modified file is placed into the repository and is used by all of the other programmers in place of the original file. The original file is retained by the configuration management tool as a version control mechanism. CMS tools provide for collaborative development, and version control of the files and the overall project, and many industries and clients require their use.

Many configuration management systems have integrated suites that include: scheduling, task assignment, defect reporting, and issue tracking systems. In addition, tools for software teams and software project managers are commonly used in industry to plan and measure project progress, and to provide visibility into the design, schedule status, cost, and quality of the code.

## Test and Integration

The next phase in the software development life-cycle is integration and testing. In the test phase, the programmer runs the program to ensure that there are no errors in the code, and that it performs correctly (meets the requirements). The two types of errors that are looked for during the test phase are syntax errors and logic errors.

*Syntax errors* - violations of language specific rules like indentation and punctuation. The compiler or interpreter will halt compilation or execution.

*Logic errors* are errors in the algorithm or the way that the algorithm was written by the programmer.

If the code is part of a larger project, it must be integrated into the overall project and tested again with the complete program. The configuration management system provides this capability as well.

## Delivery and Maintenance

The final phase of the software development life-cycle is the delivery and maintenance phase. The program is delivered to the client or customer and a period of maintaining the program begins which includes updates with added functionality, or patches that fix errors or security issues found after delivery.

# Chapter 2

## Python Language Specifics

### Comments

As mentioned previously, single-line comments begin with the pound sign in Python and will be ignored by the interpreter. The IDLE IDE used in this text will color code comments in red font as the default. For multiline comments, either three single or three double quotes at the start and end of the comment are used.

```
# This is a single line comment in Python
```

```
"""
```

```
This is a multiline comment in Python  
using three sets of double quotes
```

```
"""
```

### Displaying Output

The *print* function in Python displays output to the IDLE shell. The argument passed to the print function contains the item or items to display along with any format specifiers. Single or double quotes can be used with string arguments, and the print function automatically adds a line feed in the output. The plus sign and comma are used for multiple arguments depending on the element types.

In the examples below, note the absence of a semicolon as an end of line marker (Python does not use them), and that multiple arguments can be passed to the print function separated by commas.

```
print('Show this text.')      # displays Show this text
num = 1000                    # assigns 1000 to num
print('the number is ', num)  # displays the number is 1000
```

The *format* function is used to format output. Two arguments are passed to the function: the numeric value to be formatted and the format specification. The format function returns a string holding the formatted number that is then passed to the print function.

```
print(format(2, '.5f'))      # displays 2.00000
print(format(var, '.2f'))    # displays var with 2 decimal places
```

Multiple items passed as arguments to *print* are separated by commas. They are displayed in the order they are passed and are automatically separated in the output by a space.

```
print(5.0, 2.3, 3.4)        # displays 5.0 2.3 3.4
```

To suppress the spaces that are automatically added, pass the argument *sep= ''* to the function. This can also be used to add a specific separator.

```
print(5.0, 2.3, 3.4, sep='') # displays 5.02.33.4
print(5.0, 2.3, 3.4, sep='*') # displays 5.0*2.3*3.4
```

To suppress the line feed automatically added, pass the *end= ''* to the function.

```
print('No line feed', end=' ') # suppress line feed
print(' for these three', end=' ') # suppress line feed
print(' lines.')
```

The output from these three print statements is: *No line feed for these three lines.* The automatic line feed added by the print function is omitted.

```
No line feed for these three lines.
>>>
```



## Escape Sequences

Python's escape characters are used in quotes and include: new line '\n', tab '\t', print a single quote '\'', print a double quote '\"', and print a back slash '\\.

```
print('tab' + '\t' + 'tab')
print('extra line feed \n')
print('single quote \'')
print('double quote \'' + '\"')
print('backslash \\')
```

```
tab      tab
extra line feed

single quote '
double quote "
backslash \
>>>
```

Output

Escape Sequences

## Variables

In Python, variables are not declared by data type and can only be used if a value has been assigned to them. The single equal sign is the assignment operator and the variable receiving the value is on the left side of the operator.

*variable = expression or value*

`user_age = 29`                      *# 29 is assigned to user\_age*

The variable naming convention most used in Python is all lower case letters with multiple words separated by underscores, which aligns with the PEP 8 Python Standards Guide. Python is case-sensitive. Variable names cannot be Python key words and cannot contain spaces. The first character must be a letter or underscore and then letters, digits, or underscores can be used. Software engineering principles and most standards dictate that descriptive variable names be used to add clarity to the code.

One interesting difference in Python is that variables can reference different types while the program is running. In other words, a variable assigned to one data type can be reassigned to another type. As an example, below the variable *my\_value* is assigned 99 as an integer, is then used in an equation, and is then reassigned the string "Now a string" (single or double quotes work). The output shows the reassignment is implemented.

```

my_value = 99
print(my_value + 1)           # displays 100
my_value = 'Now a string'
print(my_value)              # displays Now a string

```

Data types categorize value in memory: *int* for integers, *float* for real numbers, and *str* is used to store strings. Floating point numbers are stored with double precision although the data type *double* is not used in Python.

## Keyboard Input

Python has a built-in function called *input* that reads input from the keyboard. The function returns the value as a string which can then be converted. The first line of code below obtains user input as a string and assigns it to *my\_string*. The second receives a value from the user and converts it and stores it as an integer in *my\_int*. The third line receives the user input and stores it as a float in *tempF*.

```

my_string = input('user prompt to obtain input')
my_int = int(input('prompt requesting an integer'))
tempF = float(input('Enter the temperature in Fahrenheit: '))

```

### Obtaining Keyboard Input

Converting an item to a different data type requires conversion similar to casting, and will only work if *item* is valid for the conversion.

```

int(item)    # converts item to an int
float(item)  # converts item to a float
str(item)    # converts item to a string

```

## Mathematical Expressions and Operators

Python operators align with other familiar languages for the most part. Addition uses (+) and subtraction (-), multiplication uses (\*), and the modulus operator is (%). The differences are that exponentiation uses two asterisks (\*\*), and there are two types of division operators. A single forward slash is used for floating point division and two forward slashes for integer division with positive results being truncated and negative results being rounded away from zero.

Operator	Description	Example	Result
+	addition	2 + 3	5
-	subtraction	72 - 12	60
*	multiplication	4 * 6	24
/	division	7 / 2	3.5
//	integer division	7 // 2	3
%	remainder (mod)	17 % 4	1
**	exponentiation	2 ** 3	8

### Mathematical Operators

#### Division Examples

```

result = 10 // 5      # result is assigned 2, an integer
result = 10 / 5      # result is assigned 2.0, a float
result = 5.5 / 5     # result is assigned 1.1, a float
result = 2.5 // 5    # result is assigned 0.0, truncated
result = 5.5 // 5    # result is assigned 1.0, truncated
result = -5.5 // 5   # result is assigned -2.0,
                    # rounded away from zero

```

For rounding numbers, Python has a *round* function that will round numbers to an integer or to a specified number of places after a comma.

```

result = round(9.4)      # result is assigned 9
result = round(9.6)      # result is assigned 10
result = round(12.2733, 2) # result is assigned 12.27
result = round(12.2755, 2) # result is assigned 12.28

```

#### Exponentiation Example

```

result = 2**4           # result is assigned 16

```

**Precedence** in Python follows PEMDAS, parenthetical expressions first, followed by exponentiation, then multiplication, division, modulo division, and lastly addition and subtraction. Operators with the same precedence are handled left to right, and precedence can be forced using parenthesis.

**Mixed-type** expression results depend upon the data types in use.

Two **int** values - the result is an **int**.

Two **float** values - the result is a **float**.

**int** and **float** - The **int** is temporarily converted to a **float**, and the result of the operation is a **float**.

Programmer type conversion of a **float** to an **int** causes truncation of the fractional part.

## The Math Module

The Python mathematical functions are contained in the math module. This module is readily available in the Python shell, but must be imported (shown later) when programming in files. The list of math functions includes: *acos(x)*, *asin(x)*, *atan(x)*, *cos(x)*, *hypot()*, *log(x)*, *sin(x)*, *sqrt(x)*, and *tan(x)*. The module also defines a value for pi and e, and provides conversions for degrees to radians, *radians(x)*, and radians to degrees, *degrees(x)*. All of these functions return float values.

When using the math functions, the math module is imported and the word math and the dot operator precede the functions as shown in these examples.

```
import math

print(math.sqrt(2))

print(math.pi * 2)      # uses 3.141592653589793

print(math.sin(30))     # value returned in radians

1.4142135623730951
6.283185307179586
-0.9880316240928618
>>>
```

## Random Numbers

Python includes random number generation with several library functions that require importing *random*. The lines below import the random library and assign *num* a random integer within the range of 1 to 100 inclusive.

```
import random

num = random.randint(1,100)
```

This line assigns a random integer from 0 thru 9 to *num*. Note that 10 is excluded.

```
num = random.randrange(10)
```

Finally this line assigns a random tenth integer from 0 to 100 to *num*, 0, 10, 20, etc.

```
num = random.randrange(0, 101, 10)
```

The random number generator can be seeded as well. When the random library is imported it uses the system time as the seed, but the following statement will provide consistent “random” numbers.

```
random.seed(10)           # any number can be used
```

For floating point random numbers between 0.0 and 1.0, the random function is used as shown here.

```
random.random()          # no arguments
```

The uniform function allows setting a range for random floating point numbers as shown here.

```
random.uniform(1.0, 10.0) # range of random floats
```

## Breaking Long Statements

Statements in Python can be broken across lines using the line continuation indicator (backslash) shown in the first example below. However, the backslash is not necessary when a statement is enclosed in parenthesis as in the second and third examples below.

```
my_variable = 35.74 + (0.6215 * tempF) - \
              (35.75 * (wind_speed**0.16))

result = (item1 + item2 + item3 +
         item4 + item5)

print ('User 1 gross pay is $', gross_pay,
      ' and the net pay is ', net_pay)
```

## Constants

Strictly speaking, CONSTANTS are not available in Python. That is, they can be changed. However, they are used and are indicated using all uppercase letters to indicate that they should not be changed.

```
EARTH_RADIUS = 3959.0
earth_circumference = 2 * 3.1415 * EARTH_RADIUS
```

## Functions and Methods - Terminology

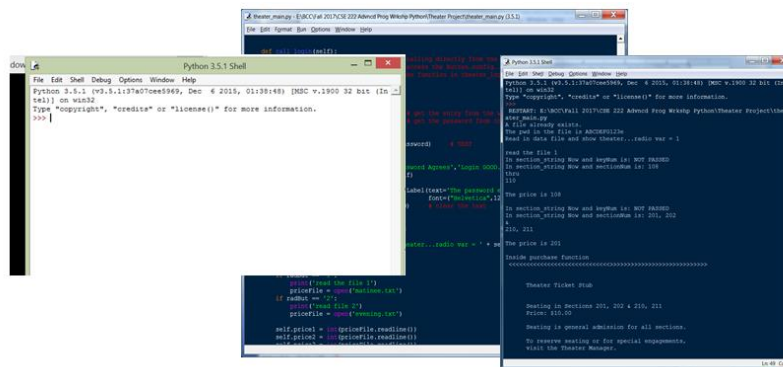
Function – a named block of executable statements

Method - a function that exists inside of an object

# Chapter 3

## Getting Started in Python

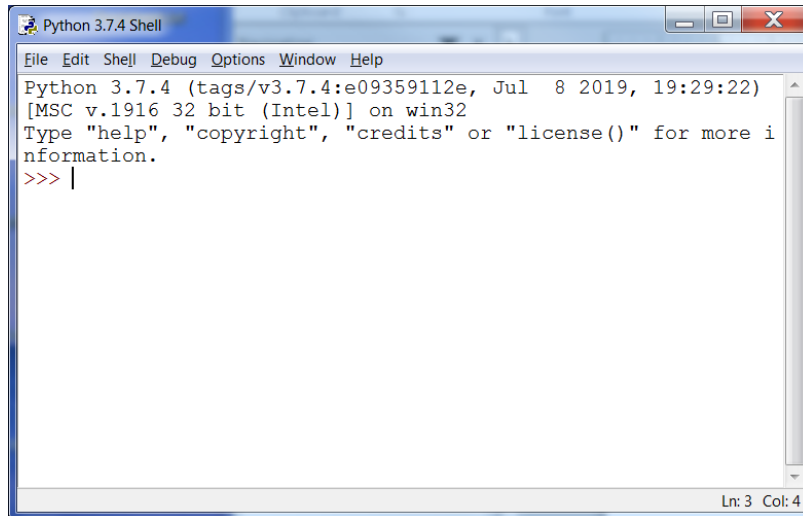
Python with IDLE is free to download and use and should be installed prior to continuing (see Appendix A). IDLE is intended to be a simple IDE that is cross-platform, and is suitable for starting out in Python especially in an educational setting. It provides a text editor and Python shell with syntax highlighting and smart indent. IDLE features an integrated debugger with breakpoint capability and call stack visibility which few students have experienced. IDLE is free to download and use (it is installed with Python), and it does not have the host of features that tend to clutter many IDEs with limited benefit. In addition, most IDEs are similar in look and feel which provides a singular exposure. The IDLE environment provides an additional educational experience in programming.



Python with IDLE

## The Python Shell and IDLE

IDLE is installed with Python and is launched using a batch file called `idle.bat` found in the `Lib/idlelib` sub-directory. Creating a shortcut at a higher level to the batch file makes launching it more convenient.

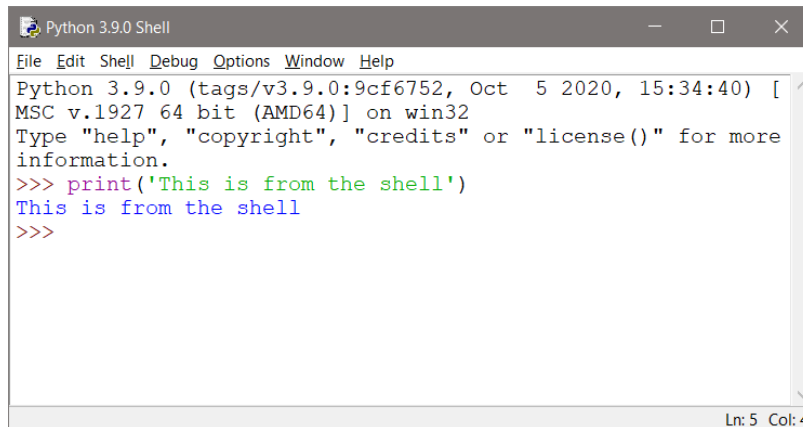


The Python Shell

Launching IDLE produces the Python shell. Single lines of code can be written directly into the shell and run there, and help is available by typing `help()`.

**Ex. 3.1** - A line of code entered into the shell will execute when *Enter* is pressed.

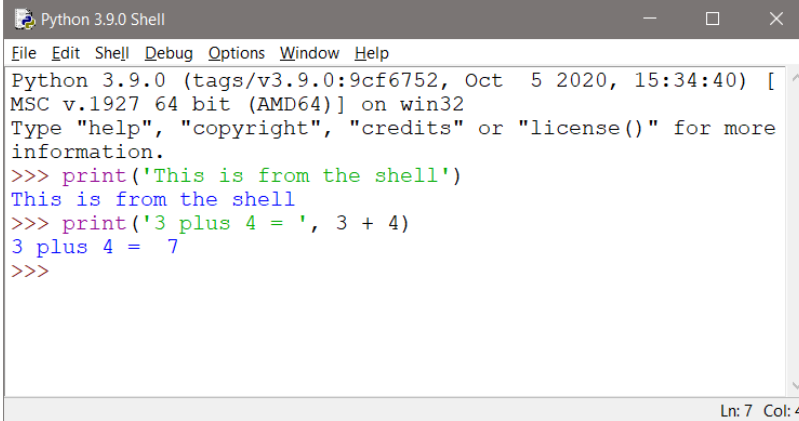
```
print("This is from the shell")
```





Ex. 3.2 – This line of code includes an equation as the second argument passed to the print function. Both are output when Enter is pressed.

```
print("3 plus 4 = ", 3 + 4)
```

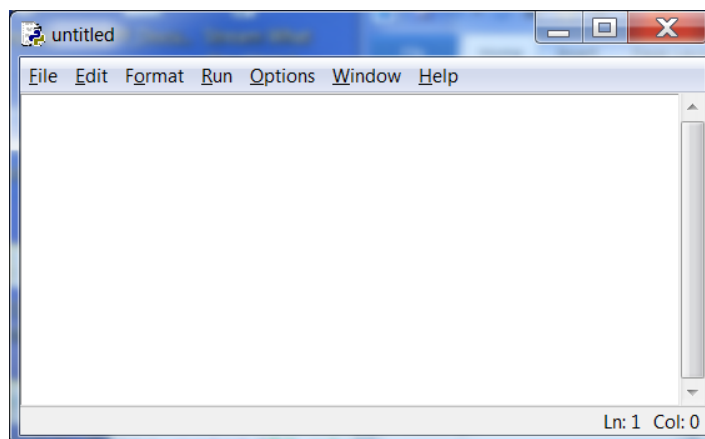


```
Python 3.9.0 Shell
File Edit Shell Debug Options Window Help
Python 3.9.0 (tags/v3.9.0:9cf6752, Oct 5 2020, 15:34:40) [
MSC v.1927 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more
information.
>>> print('This is from the shell')
This is from the shell
>>> print('3 plus 4 = ', 3 + 4)
3 plus 4 = 7
>>>
```

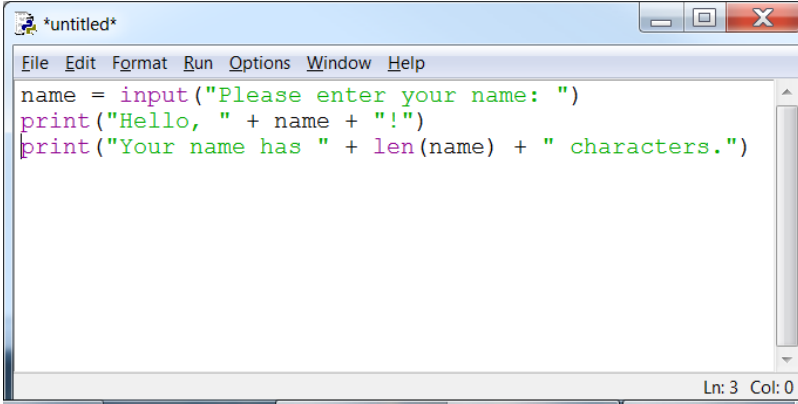
Programming in the shell and executing the lines works well for code snippets or examples, but the goal is to write complete Python programs. The shell simply uses the interpreter to execute the lines we've typed when *Enter* is pressed. Files will be used to write and execute more complex programs.

Ex. 3.3 – The IDLE editor is started by choosing *File -> New File* from the menu.

The new window is the edit window where sequences of Python commands are entered. Unlike the shell where the lines execute when Enter is pressed, the lines in the editor will be executed as a group to form a program. The title of the window below will change from “untitled” when it is saved. The drop-down menus provide basic IDE functionality.



Ex. 3.4 – The lines of code in the edit window below have an intentional error. Running the lines shown will reveal one way that IDLE indicates errors.



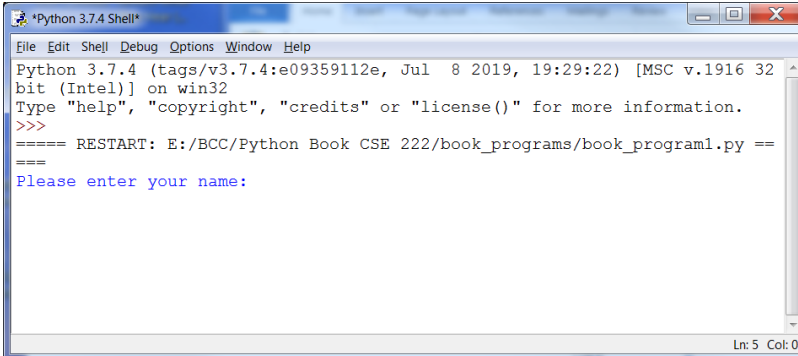
```

File Edit Format Run Options Window Help
name = input("Please enter your name: ")
print("Hello, " + name + "!")
print("Your name has " + len(name) + " characters.")
Ln: 3 Col: 0

```

New File Window

To run the program, select “Run” and then “Run Module” from the menu or just press F5. Either way, IDLE will force saving the file before running the program. After the file has been saved, Python will run the program.



```

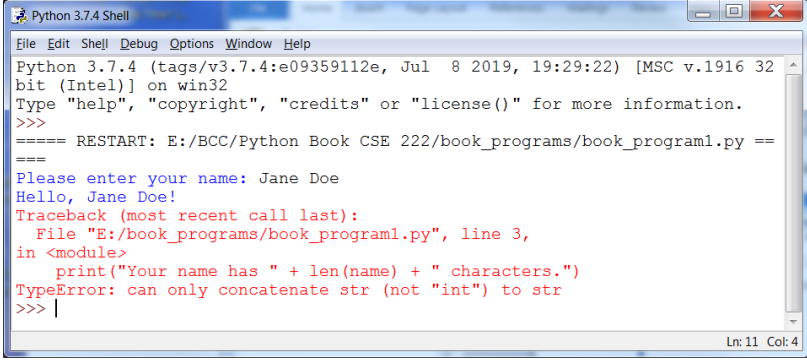
Python 3.7.4 Shell
File Edit Shell Debug Options Window Help
Python 3.7.4 (tags/v3.7.4:e09359112e, Jul 8 2019, 19:29:22) [MSC v.1916 32
bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: E:/BCC/Python Book CSE 222/book_programs/book_program1.py ==
=====
Please enter your name:
Ln: 5 Col: 0

```

Program Running in the Python Shell

When a response to the prompt is entered, the intentional error in the code will surface. The error information provided includes the file name and line number for the error, as well as the line of code itself along with the type of error. The line numbers in the program can be seen at the bottom right of the Python program window. The function call `len(name)` returns an integer which is the

length of the string passed to it in *name*. Python cannot concatenate an integer onto the string “Your name has “, so execution stops.



```
Python 3.7.4 Shell
File Edit Shell Debug Options Window Help
Python 3.7.4 (tags/v3.7.4:e09359112e, Jul 8 2019, 19:29:22) [MSC v.1916 32
bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: E:/BCC/Python Book CSE 222/book_programs/book_program1.py ==
=====
Please enter your name: Jane Doe
Hello, Jane Doe!
Traceback (most recent call last):
  File "E:/book_programs/book_program1.py", line 3,
in <module>
    print("Your name has " + len(name) + " characters.")
TypeError: can only concatenate str (not "int") to str
>>> |
```

### Traceback Error

To correct this, convert the result of the function call to `len` to a string using `str` as shown below.

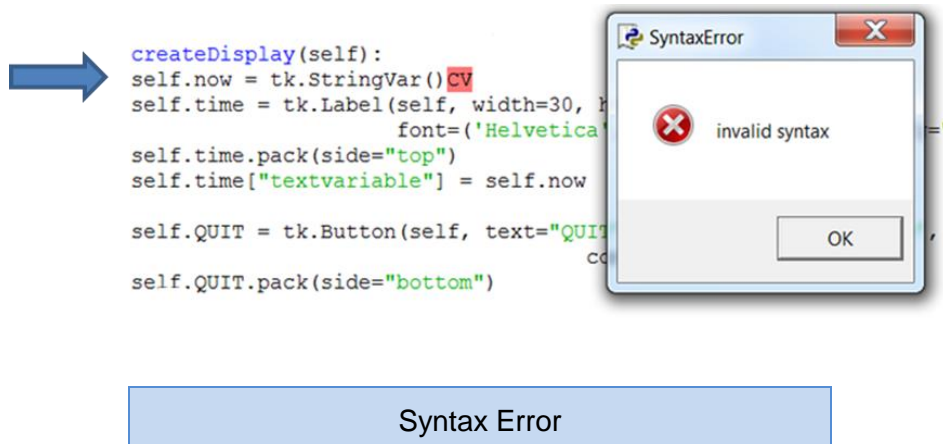
```
name = input("Please enter your name: ")
print("Hello, " + name + "!")
print("Your name has " + str(len(name)) + " characters")
```

**Ex. 3.5** – After correcting the code, saving it, and running it again (F5), the program now runs correctly (notice that the space was counted by `len`).

```
Please enter your name: Jane Doe
Hello, Jane Doe!
Your name has 8 characters.
>>> |
```

That is one type of error alert in IDLE. Another type is a syntax error. In many cases IDLE will highlight the actual code where the error occurs by boxing it in red and producing an error dialog.

In the example below, the characters “CV” were erroneously typed at the end of the second line. The actual error in the code is highlighted and IDLE produced a syntax error dialog box.



## Exiting Python

To leave IDLE, just close the windows.

Since IDLE insists that files are saved before each execution, it's hard to lose changes when exiting IDLE.

To be really safe, save the program manually before closing the editing window.

Choose "File" on the menu bar and "Save" from the drop-down menu or use Control-S.

# Chapter 4

## Decisions, Logic, Loops, and Functions

### If, else, and elif

The syntax for the **IF** statement has three distinctions in Python; first parenthesis do not surround the conditional statement, second, a colon follows the conditional statement, and third, a block of code associated with it is formed through indentation, not braces. The general format for an “if” is shown below.

```
if conditon:  
    statement1  
    statement2  
    statement3  
    etc.
```

The **ELSE** condition is handled the same way.

```
if condition:  
    statments  
else:  
    statements
```

The else condition also requires a colon, and the IF clause and the ELSE clause must be aligned.

For an else-if condition, Python uses **ELIF** as shown below. Again, the clauses are aligned using indentation.

```

if condition_1:
    statement(s)
elif condition_2:
    statement(s)
elif condition_3:
    statement(s)
else:
    statement(s)

```

## Boolean Logic and Relational Operators

Boolean Logic and relational operators in Python are similar to other languages, and resolve to either True or False. The operators function as follows:

- > greater than
- < less than
- >= greater than or equal to
- <= less than or equal to
- == equivalent (two equal signs without a space)
- != not equivalent

### Relational Operators

Strings can be compared using the equivalence operator which compares each character based on their ASCII values.

```

if string1 == string2:           # comparing strings
    print('They are equal')

```

The logical operators are the actual words “*and*”, “*or*”, and “*not*”, and the IDE will color code these for clarity. Short circuit evaluation is also used as in other languages; meaning, in a logical *and* condition, if the left expression is false, the right expression is not evaluated. In a logical *or* condition, if the left expression is true, the right expression is not evaluated.

### Logic Operator Example

```
value > 0 and value < 20      # Logical AND
value < 0 or value > 100     # Logical OR
```

Boolean variables are also available in Python as the `bool` data type which operates as true or false.

### Repetition Structures (Loops)

Repetition structures follow the colon and indentation rules associated with conditions. A colon is placed after the condition and indentation forms the block of code executed when the condition is true. A **WHILE** example follows:

```
while condition:
    statement1
    statement2
    etc.
```

The Python **FOR** loop has some differences that require explanation. It is designed to work with sequences of data and iterates once for each item in the sequence. Each value in the brackets will be placed in *variable* for use in the loop statement or statements.

```
for variable in [item1, item2, item3, etc.]:
    statement1
    statement2
    etc.
```

The next few examples are in exercise format to provide some experience with Python loops. To execute them, launch IDLE, open a file, and enter the example code. Save the file each time and press F5 to run.

#### Ex. 4.1 – FOR loop example

```
for temp in "something":
    print(temp)      # displays each letter
```

When this code is run, each letter in the word “something” is placed in the variable *temp* one letter at a time and is then passed to the *print* function. The letters are displayed vertically because the print function adds a line feed.

## Range

Python has a **RANGE** function to simplify writing limited and count-controlled loops. The function can accept one, two, or three arguments as shown below.

When **one argument** is passed to the *range* function, it is used as an ending limit for the range beginning at zero.

### Ex. 4.2 – FOR loop using **Range** with a **Single Argument**

```
for value in range(100):    # prints 0 through 99
    print(value)           # 100 is not output
```

When the code executes, the output begins at 0 and ends at 99 since 100 is the limit and it is not included.

When **two arguments** are passed, they are used as the starting and ending limits of the series.

### Ex. 4.3 – FOR loop using **Range** with **Two Arguments**

```
for value in range(20, 50): # prints 20 through 49
    print(value)
```

The output for this example begins at 20 and ends at 49 since 50 is the limit.

When **three arguments** are passed the third argument is used as the step value for the series.

### Ex. 4.4 – FOR loop using **Range** with **Three Arguments**

```
for value in range(20, 50, 5): # prints 20, 25, 30, 35, 40, 45
    print(value)
```

The outputs for this loop are multiples of 5 from 20 thru 45 since the third argument is the step.

Any or all of the integer literals in the range function examples above can be replaced with variables. In the next example, all three literals are replaced with variables.



Ex. 4.5 – Range function using three variables instead of integer literals

```

first = 24
second = 49
third = 6

for value in range(first, second, third):
    print(value, end=' ')      # displays 24 30 36 42 48

```

## While Loop

The while loop structure is similar to other conditional structures with a colon after the condition and indentation for the statements associated with the loop. The general format is:

```

while condition:
    statement1
    statement2
    etc.

```

There is not a do-while loop construct in Python. A while loop or for loop is used to implement the logic required. A while loop example follows.

```

index = 0
while index < 5:
    print(index)          # displays 0,1,2,3,4
    index += 1

```

## Functions

There are two types of functions in Python, *void functions* that just perform a task and *value-returning functions* that return a value. The code for a function in Python is called the *function definition* and it begins with the keyword *def* which is followed by the name of the function, a pair of parentheses, and a colon. The first line is referred to as the *function header*, and the statements that will execute when the function is called are indented and form a block of code (the function body). The general format is shown below.

```

def function_name():
    statement1
    statement2
    etc.

```

A reminder about *indentation* is warranted. Indentation forms a block of code in Python. The function names, including main begin at the margin, and the function bodies are indented forming a block of code for the function. The IDE highlights items by color-coding the text as shown below. Also note that it is much easier to use the tab key for indentation than to count spaces to be sure they are always the same.

```
def main():  
    function1()  
    function2()  
    function3()  
  
def function1():  
    print('F1')  
  
def function2():  
    print('F2')  
  
def function3():  
    print('F3')  
  
main()
```

Function Structure

## Function Variables and Scope

When a variable is declared within a function, its scope is the function (this includes the main function), and it is referred to as a *local variable*. Therefore, a variable defined inside a function is not accessible outside that function, and different functions could have variables with the same name without causing any conflict. Each of the variables would have its particular function as its scope, and would not be accessible by the other function. If several engineers are working on the same program, but they are working on different functions, they may name a local variable using the same name.

In the examples that follow, line numbers are shown for the explanations that follow. Displaying line numbers is available in the IDLE Options menu.

In Example 4.6, Line 3 defines and names the function using *def*, the name of the function, a pair of parenthesis, and a colon. Note the indentation of lines 4 and 5 which form the block of code associated with function. Line 8 is the actual function call. Notice that the call to the function does not have a colon.

As the interpreter reads through the code, it executes the function when line 8 is reached. The example program simply executes the function. There is no main function in this example, but it will run.

#### Ex. 4.6 – A Simple **Function**

---

```

1 # A simple function example
2
3 def output_function():
4     print ('This function simply prints output ')
5     print ('when it is called on line 8.')
6
7 # The function is called
8 output_function()
9

```

Adding a main function provides flow of control.

#### Ex. 4.7 – A simple **Function** called from **main**

---

```

1 # This program has a main function that calls
2 # the output_function
3
4 def main():
5     print ('Now inside main')
6     output_function()
7     print ('Back in main')
8
9 def output_function():
10    print ('This function simply prints output')
11    print ('when it is called.')
12
13
14 # Call the main function
15 main()

```

Notice that the function definition starting on line 9 seems to be inside main, but it is below main and out-dented the same as main. Line 15 is the call to main which begins execution at line 4. The output function is called on line 6 and

executes. As many functions as are needed can be added. Recall that the interpreter reads through the lines top to bottom. When it reaches line 15 it executes the program. The output is shown below.

```
==== RESTART: E:/output_function.py ====
Now inside main
This function simply prints output
when it is called.
Back in main
>>>
```

## Functions, Passing Arguments

Passing **arguments** to functions are handled similar to other languages with the exception of a data type in the function header. Like Java, there is no pass-by-reference (Python allows multiple values to be returned by functions which will be covered later). The previous program has been modified below to pass an argument. Main assigns a string to the variable on line 5, and passes the variable on line 6 to the function. The function has been modified on line 9 to accept the variable, and line 10 prints the phrase.

### Ex. 4.8 – Passing an Argument from main to a function

```
1 # Passes an argument to output_function from main
2
3 def main():
4     print ('Now inside main')
5     phrase = 'arg1'
6     output_function(phrase)
7     print ('Back in main')
8
9 def output_function(item):
10     print(item + ' was passed in.')
11
12
13 # Call the main function
14 main()
-
```

```
Now inside main
arg1 was passed in.
Back in main
>>> .
```

When passing multiple arguments to functions in Python, the ordering of the arguments must be consistent between those passed and those received. However, Python does allow re-ordering arguments when passing them using *keyword arguments* which are covered next.

Keyword Arguments specify to the receiving function which parameter is to receive which value that is being passed. As an example, the function below has three arguments: first, second, and third. The call to the function passes them in the incorrect order but specifies the parameter and value.

#### Ex. 4.9 – Passing Keyword Arguments from main to a function

```

1 # Passes key word arguments from main to a function.
2 def main():
3     output_function(third=3, second=2, first=1)
4
5 def output_function(first, second, third):
6     print (first, second, third)
7
8 # Call the main function
9 main()
10

```

```

1 2 3
>>>

```

The output is 1 2 3 because the function correctly assigns the values. The print function puts a space between each of the variables being displayed by default.

### Functions, Returning Multiple Values

As mentioned previously, functions in Python can return multiple values. This resolves some of the issues created by not having the ability to use pass-by-reference which would allow a function to change multiple variables permanently. Recall that pass-by-reference allows a function to change the value of a variable passed in as an argument because the function has a reference to the variable - access to the memory location of the variable.

The following example includes a function returning a single value and a second function that returns multiple values. The returned values must be received in the order that they are returned.

Notice that all of the function definitions (lines beginning with `def`) are at the margin. `main` is a function as is `get_sum` and `get_sum_and_dif`. They are each defined and when `main` is called on the last line, the program executes and calls the functions.

**Ex. 4.10** – Returning Single and Multiple Values from a function.

```
# This program returns a single and then multiple
# values from functions called from main
def main():
    sum = get_sum(5,3)           # call get_sum
    print (sum)

    sum, dif = get_sum_and_dif(10,4) # call get_sum_and_dif
    print (sum, dif)

def get_sum (first, second):    # returns a single value
    return first + second

def get_sum_and_dif (first, second): # returns two values
    sum = first + second
    dif = first - second
    return sum, dif

main()                          # call the main function

8
14 6
>>>
```

To return a **Boolean** value, assign `true` or `false` to a variable and return it. A function can also simply return `true` or `false` if there is an assignment statement to receive it.

```
if condition:
    my_bool = True
    return my_bool
else:
    my_bool = False
    return my_bool
```

As shown later in the text, functions defined in other files require the file name and the dot operator to access.

## Global Variables

Global variables should be used sparingly if at all since their use makes debugging very difficult. Declaring a variable outside any function in Python makes it global and accessible to all areas of the program. Then to assign a value to a global variable inside a function requires the global keyword to precede the variable.

```
num = 1                                # global variable declared

def main():
    global num                          # global key word assigned

    print('Num is now ', num)

    num = int(input('Integer: '))      # assign new value

    show_num()                          # function called with no arguments

def show_num():
    print('In show_num, and num is ', num)

main()

    Num is now 1
    Integer: 44
    In show_num, and num is 44
    >>>
```

Note that in the program above, **num** must be declared as global in main using the keyword prior to an assignment statement. Combining them as shown here is not permitted.

```
global num = int(input('Enter a number: ')) # invalid syntax
```

## Pointers

The Python language does not have pointers per se, but object references. When a string (string object) is created, there is a reference in memory. Strings are immutable but when a string is modified in Python, a new string is created and the reference for the original string variable is now a reference to the new string memory location. The interpreter will eventually remove the original now-unreferenced original string variable.

The following program reveals memory locations using the *id()* function which returns the identity of an object. The program then shows the different memory locations used by passing an argument and by modifying a string.

```
def main():
    my_str = 'string'
    print('\n Memory 1 is: ', id(my_str))
    output_function(my_str)
    print('\n Memory 4 is: ', id(my_str))
    my_str = my_str + ' more'
    print('\n Memory 5 is: ', id(my_str))

def output_function(the_str):
    print('\n Memory 2 is: ', id(the_str))
    the_str = the_str + 'less'
    print('\n Memory 3 is: ', id(the_str))

main() # Call the main function
```

The output for the program is:

```
Memory 1 is: 2902688          # initial string
Memory 2 is: 2902688          # received in the function
Memory 3 is: 47600688        # modified string in the function
Memory 4 is: 2902688          # back in main
Memory 5 is: 47600688        # modified string in main
>>>
```



# Chapter 5

## GUI Design & Development

Graphical User Interfaces are event driven by user input. The user determines the sequence of many of the events; therefore, careful design is required to control access to the events. For instance, a button that computes a result that requires user input of a value should not be enabled until the user has input the required value. Situations like this must be considered in the design phase of the interface, otherwise required modifications will surface in the test phase when issues are found. This increases the input validation aspects of the program. A value needed for computation must be entered by the user before allowing computation, and the value entered must be within the correct range of values for the computation to avoid issues such as division by zero.

Consider a program that computes the circumference of a circle based on an input of radius.

1. The radius must be input prior to computation
2. The radius input by the user must be a number
3. The radius input by the user must be a positive number

The graceful handling of incorrect input values is required for a robust and well-engineered solution. In a non-GUI program, we might use a loop that iterates until a correct value is entered. It would display an error message to alert the user to the issue and re-prompt for input inside the loop. The same concept holds true for a GUI program, but with the added requirement of employing

windows to handle the tasks. This situation will be explored later. Generating the main GUI is the first step. This requires generating a window and placing controls (aka widgets or components) on it. A control is an element of the interface that enables a user to accomplish some function or to access an area of the program.

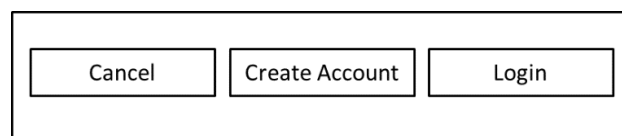
Python is a well suited language for creating graphical user interfaces through the Tkinter module. The module is installed with Python and provides windows and controls that are easy to program. The Tkinter package is the standard Python interface into TK GUI Toolkit. The name Tkinter is short for TK Interface and the TK Toolkit is used by many developers in other languages as well.

Some of the Tkinter controls include:

Button	causes an action or event when clicked
Canvas	rectangular area for graphics
Checkbutton	On or Off position check boxes
Entry	single line entry control
Frame	container that can hold controls
Label	area that displays one line of text
Listbox	user selection list
Menu	list exposed when a menu button is clicked
Radiobutton	select/deselect control

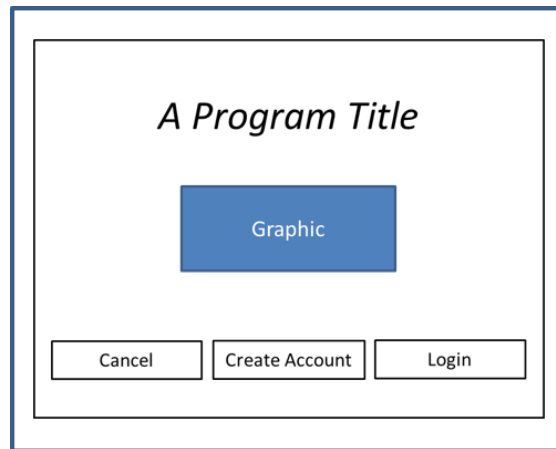
Before selecting controls for the interface, a preliminary design should be completed. This provides a layout for the window and an idea of how it will look and operate prior to writing any code. Storyboarding (walking through the program operation steps) can also be helpful at this stage. The example in this and subsequent chapters builds the Data Analysis Project in Appendix C.

The example project requires a standard initial window with three buttons: Login, Create Account, and Cancel. This is the main window for the program and entry point for the user. A first sketch might simply include a window and the three buttons.



The opening or initial window is the first impression of the program for the user. An improved sketch might include a program title and a graphic that reflects the nature of the program.

**Ex. 5.1** – Example Sketch of initial GUI.



Initial GUI Sketch

To create a main GUI with a title, a graphic, and three buttons, Python provides controls and layout managers including: the Place Geometry Manager, Grid Geometry Manager (grid), and the Pack Method to position the controls. Additional layout managers are available in modules that can be added to Python using the PIP installer shown in Appendix B. The examples will use grid and the Tkinter package which is installed with Python. Controls can be located with grid using a two-dimensional table of rows and columns.

## Generating the Main Window

Creating the main window requires introducing additional components, and the Python *main loop*. The tkinter main loop executes when the program starts and continues to run waiting for user actions until the user ends the program. This has many benefits and adds a few additional considerations.

Programmers use an object oriented approach to GUI development, and the main window will be created as an instance of the main window class.

*Tip: Line numbers for the code are shown in the bottom right corner of the IDLE Editor.*

### Ex. 5.2 – Main GUI, Interface Code.

As noted previously, attention to indentation levels is critical in Python. The code below generates a window (*main\_win*), having a minimum size, title, and label.

```

1. import tkinter as tk
2.
3. class DataGUI:
4.     def __init__(self):
5.         self.main_win = tk.Tk()
6.         self.main_win.title("Data Analysis Example")
7.         self.main_win.minsize(width=550,height=200)
8.
9.         self.heading_label = tk.Label(text='Data Analysis Program',\
10.                                     font=("Helvetica",16), fg="blue")
11.         self.heading_label.grid(row=2, column=1)
12.
13.         tk.mainloop()
14.
15. dataAnalysis = DataGUI()

```

Line 1 imports the Tkinter module as tk which allows using tk when accessing the library as shown on line 11 where the tk.Label is created.

Line 3 declares the class.

Line 4 begins the initialization function for the window with two underscores before and after the word init. Self refers to the newly created object.

Line 5 creates the main window.

Line 6 adds a title to the window border. The icon on the border can also be changed as we'll see later.

Line 7 sets a minimum size for the window.

Lines 9 & 10 create a label using the text, font style and size options.

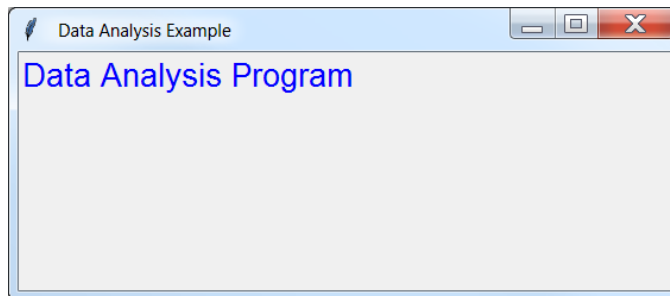
Line 11 positions the label created on line 9 using grid geometry.

Line 13 starts the tk main loop.

Line 15 creates an instance of the DataGUI class called dataAnalysis.

In Ex 5.2, the image and buttons have not been included. It simply creates the main window with a title on the title bar, and a heading label. In software engineering, building a portion of the project and testing that portion before moving on is referred to as the “build a little, test a little” approach. As new code is added, any issues or errors that surface would be in the added code. Is easier to debug five lines of code or fifty lines of code?

The Main GUI code in Ex. 5.2 runs and produces this window.



Simple Window with Title and Label

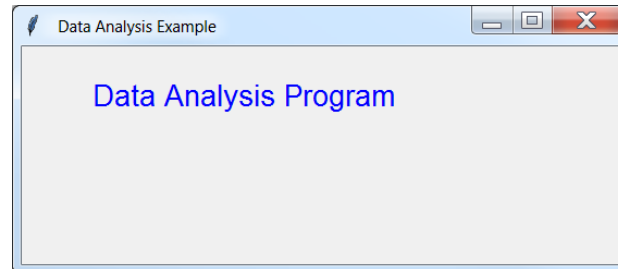
The code when executed will generate the window. The window title is on the border, and the program title (*heading\_label*) is Helvetica, font size 16, and blue, but the label is positioned far left even though the grid positions of row 1 and column 2 were designated (copied below).

```
self.heading_label = tk.Label(text='Data Analysis Program',\
                             font=("Helvetica",16), fg="blue")
self.heading_label.grid(row=2, column=1)
```

The *Grid Geometry Manager* (grid) will size each row and column to the smallest size required for the items positioned in them. In this case there is nothing in row 0 or 1 and nothing in column 0, so the position of the program title doesn't display where the code indicated. It might be tempting to place *invisible* controls in rows and columns to fill them with “something” so that the controls are forced to be in the desired positions. Here is an example using a blank label to try and move the program title. Notice that the text for the label is simply spaces between quotes.

```
self.main_win.blank_label = tk.Label(text="          ") # blank label
self.main_win.blank_label.grid(row=0, column=0)
```

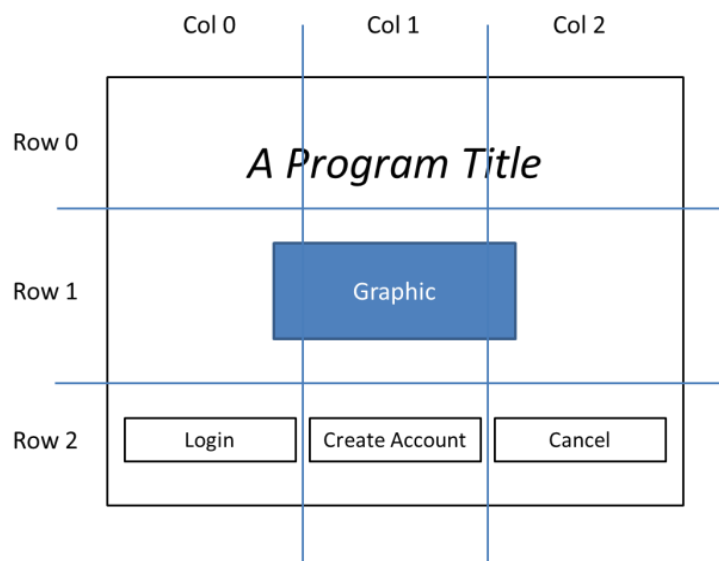
As shown below, placing the blank label in row 0 column 0 moves the title down and to the right, but not to the desired location. There is also no indication as to how this would affect the other controls, especially, the Login button, which in the design is on the left side of the window and will either be in column 0 or 1.



Example with Blank Label Added

### Positioning Controls (GUI Design)

The positioning of elements on the window is part of the design phase. The original sketch of the main window in the example included the program title, image, and three buttons. Since the grid manager positions elements by row and column, adding lines to the preliminary sketch provides a better representation of the main window and where elements would be located.



Modified Sketch with Grid Lines

The modified sketch that includes lines for the rows and columns indicates: the program title is located in row 0 column 1, the image is in row 1 column 1, the Login button is in row 2 column 0, the Create Account button is in row 2 column 1, and the Cancel button is in Row 2 column 2. Notice that the “Program Title” and the image go beyond the boundaries of the columns they occupy. The different sizes of the elements must be considered when positioning them with the grid layout and for the options used. The program title and the image span 3 columns, and there is a grid option called *columnspan* that can resolve this. For the rows, if the buttons are considered to be a single row, then the title and image also span 2 rows in height. The Grid option *rowspan* can resolve this.

The options *rowspan* and *columnspan* along with others provide flexibility in the design and positioning of the elements.

Grid positioning options include:

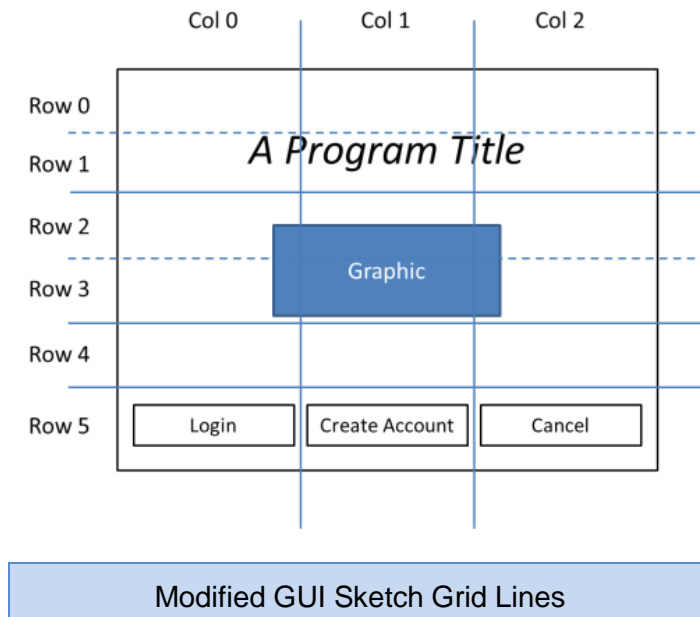
column	column location of the control
columnspan	allow a control to span multiple columns
ipadx	horizontal padding within the control borders
ipady	vertical padding within the control borders
padx	horizontal padding around the control within a cell
pady	vertical padding around the control within a cell
row	placement of the control on the grid
rowspan	allow a control to span multiple rows
sticky	One or more of N, S, E, W to align controls within cells

Planning the control positions and considering the use of options can save a lot of time adjusting after the fact. Adding padding or a span option to one control can move others which then forces changes to them as well. Then adjusting the options for those controls can counteract the original change or create more needed adjustments.

Focusing on the smallest or largest control in a row or column is an approach that can save time and minimize adjustments. Consider in the example that the buttons will be the same width and height and that they are the smallest controls in the GUI. The other controls could be “spanned” to accommodate the row and

column sizes used for them. A redrawn main window sketch with modifications to accommodate spanning columns and rows might appear like this:

Ex. 5.3 – Main GUI sketch redrawn.



From this sketch, positioning and spanning can be determined, but there are also methods to configure columns to give them a minimum width and to configure rows for a minimum height. If a control is the sole occupant of a row or column, the row or column size can be increased. That way the control does not have to span multiple rows or columns. This is helpful, and it is important to remember that all rows do not need to be the same height, and that all columns do not all need to be the same width.

Ex. 5.4 – Column and Row Configuration Settings

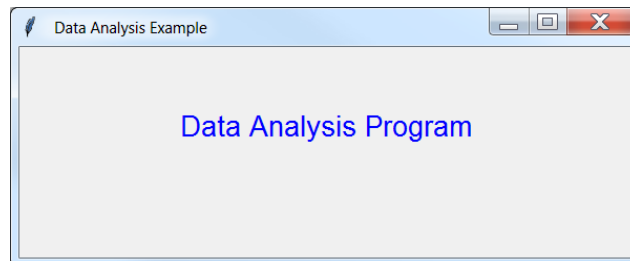
```
self.main_win.columnconfigure(0, minsize = 150) # These 3 lines set
self.main_win.columnconfigure(1, minsize = 175) # the column widths
self.main_win.columnconfigure(2, minsize = 150) # for the GUI
self.main_win.rowconfigure(0, minsize = 50)      # moves title down

self.heading_label = tk.Label(text='Data Analysis Program',\
                               font=("Helvetica",16), fg="blue")
self.heading_label.grid(row=2, column=1)
```

Adding configuration settings for the columns accommodates and centers the label horizontally. The configuration setting `minsize=50` for row 0 moves the



program title down, and adding the *rowspan* option to the grid positioning of the label accommodates the height of the font. Note that this could also be done by setting a minimum row height for row 2 where the label is positioned. The settings can be modified to accommodate design and preference.



Configured and Aligned Simple Window

## Button Controls

GUIs typically contain button controls that allow some action to take place when users click them. The Tkinter button control has many options for customization and can be positioned much the same as a label.

Button options include:

<code>bg</code>	background color
<code>fg</code>	foreground (text) color
<code>font</code>	text font for the button face
<code>height</code>	height of the button in text lines (font dependent)
<code>image</code>	image to be displayed on the button
<code>justify</code>	multiple text line alignment (LEFT, CENTER, RIGHT)
<code>padx</code>	padding left and right of text
<code>pady</code>	padding above and below text
<code>relief</code>	type of border: SUNKEN, RAISED, GROOVE, and RIDGE
<code>state</code>	enable or disable the button (normal, active, disabled)
<code>width</code>	the width of the button

Performing a control review at [Python.org](http://Python.org) would provide many other options for buttons, a few of which will be used and explained in subsequent examples.

The GUI in the example requires three buttons: Login, Create Account, and Cancel. From the design sketch, they occupy columns 0, 1, and 2 respectively. The Login and Create Account buttons will call functions in the program for operations and the Cancel button ends the program. The command option for the Cancel button shown in Ex. 5.5 below *destroys* the window and ends the program.

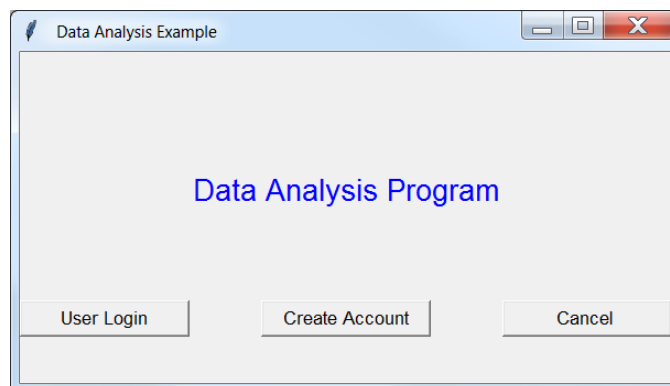
The syntax for buttons follows a *name (Master, options)* convention. The example below omits “*self.main\_window*” as the Master, since it is implied. The first option is the text to appear on the button (in this case “Cancel”), followed by a width setting for the button, the font for the text, and the command to execute when the button is clicked (the *callback* function). The button creation is followed by a line of code that positions the button on the grid.

#### Ex. 5.5 – Cancel Button Creation and Placement

```
self.quit_button = tk.Button(text=' Cancel ', width=16, \
                             font=("Helvetica",10), \
                             command=self.main_window.destroy)
self.quit_button.grid(row=4,column=2)
```

The width setting for the Cancel button was chosen based upon the length of the label for Create Account which is the longest button label. The Create Account button would be wider than the other buttons if they weren’t adjusted. The other buttons are handled similarly, but without the command option for the functions. This allows testing the creation and positioning of the buttons.

#### Ex. 5.6 – Button Creation and Positioning.



Configured Window with Buttons

For the window in Ex. 5.6, a few additional adjustments were made to the code for the other two buttons. The main window *minsize* option was increased in height, and a *rowconfigure* for row 3 was added. Row 3 will contain the image in the next step, so an adjustment was needed. Note that the example used row 4 as the row for the buttons.

The example's main GUI is developing, but the buttons are placed at the left and right edges, and the image hasn't been added. Adjusting the buttons could be done with *padx*, which would put space on the left and right of the buttons or columns could be added to the sides of the window.

#### Ex. 5.7 – Using *padx* with Button Controls

```
self.quit_button = tk.Button(text='  Cancel  ', width=16, \
                             font=("Helvetica",10), \
                             command=self.main_window.destroy)
self.quit_button.grid(row=4, column=2, padx=15)
```

## Images

Adding the image to the main window will be similar to positioning the controls. Python's Tkinter has a *PhotoImage* class for handling images that supports the GIF and PGM/PPM formats. If other file formats are needed, the Python Image Library (PIL) contains classes that can handle over 30 formats and convert them to Tkinter compatible image objects. The image file can be located with the program files, which is the default directory, or a path to the file can be used.

When a *PhotoImage* instance is used, a reference to the image must be retained or Python's interpreter could eliminate it even if it is being displayed. Also, to use *PhotoImage*, the entire Tkinter module must be imported. To do this, a line of code is added above the import for tkinter.

```
from tkinter import *      # imports everything
import tkinter as tk      # imports tkinter as tk
```

Note: Avoid using *wildcard import statements* when multiple modules are imported. Name clashes can occur when modules have functions or classes with the same name.

## Adding the Image

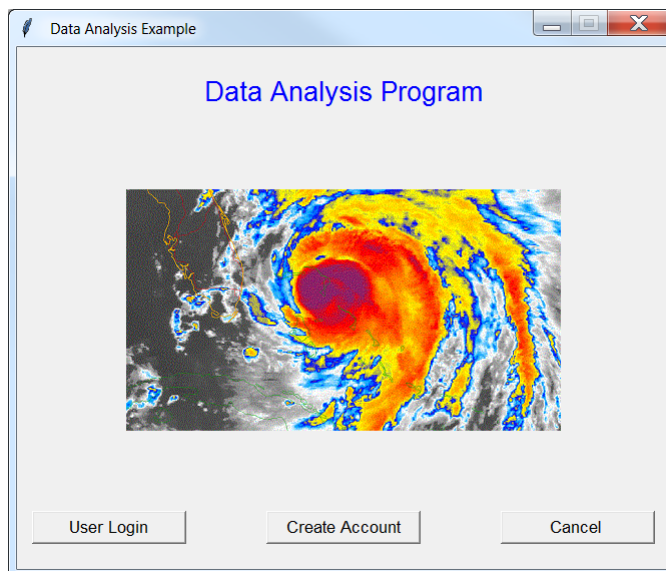
The code to use an image consists of four lines. The first line assigns the file to a PhotoImage object, the second places the image on a label (a canvas or frame can also be used), the third retains a reference to the image, and the fourth positions the image in the grid.

### Ex. 5.8 – Adding an Image

```
photo = PhotoImage(file= "Storm_image.gif")
self.labelGIF = tk.Label(image= photo)
self.labelGIF.image = photo      # retain a reference
self.labelGIF.grid(row=3, column=0, columnspan=3)
```

Note the use of columnspan for the image as well as some adjustments needed to other controls. For the resulting GUI shown below, the buttons were moved down a row and row 6 was added with the rowconfigure minsize option to add some space below the buttons. The rowconfigure minsize for row 3 was also increased to accommodate the image. Since no other control is in the row, configure was used instead of rowspan.

### Ex. 5.9 – Main Interface.



Completed Main Interface Window

The example project buttons are inactive with the exception of the Cancel button which destroys the window and ends the program. The functionality for the others will require additional design considerations including entry controls and file handling which is covered in the next chapter.

## Freezing Window Size

The controls are positioned for a window of a specific height and width set in the program. If a user stretches the window in any direction, the controls will no longer be in those positions. To keep this from happening, the `resizable` function for height and width can be used to set them to `False`. There are two variations.

```
self.resizable(height = False, width = False)
self.resizable(False, False)
```

## Frames and LabelFrames

The *frame* container is a rectangular area that can be used for padding in a window or to group controls when positioning complex layouts. It can also act as a placeholder for video inserts.

Frame options include:

<code>bd</code>	size of the border (defaults to 2 pixels)
<code>bg</code>	background color
<code>height</code>	height of the frame
<code>relief</code>	type of border: flat, groove, raised, ridge, solid, or sunken
<code>width</code>	width of the frame

The default relief is flat. To use the other relief options the border size must be increased using the *bd* option. The default size for a row and column is 1 pixel, so they must be resized to accommodate the frame. The following code generates three frames of different colors in a window. Note the use of loops to give the rows and columns *weight* using *configure*. *Weight* is used to distribute added space between rows and columns. A row or column with the weight 2 will grow twice as fast as one with weight of one. The default is zero (it will not grow at all).

### Ex. 5.10 – Frame Container Example

```
class FrameGUI:
    def __init__(self):
        self.main_win = tk.Tk()
        self.main_win.title("Frame demo")
        self.main_win.minsize(width=400,height=400) # window size

        for r in range(6):
            self.main_win.rowconfigure(r, weight=1)
        for c in range(5):
            self.main_win.columnconfigure(c, weight=1)

        self.main_win.Frame1 = tk.Frame(self.main_win, width=50, bd=5,\
            relief='raised', height=40,bg="red")
        self.main_win.Frame1.grid(row = 0, column = 0)

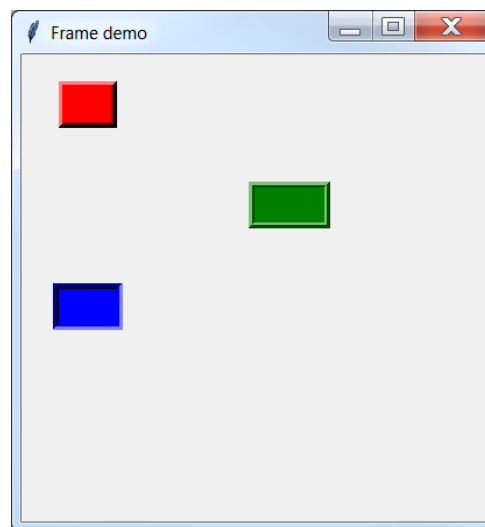
        self.main_win.Frame2 = tk.Frame(self.main_win, width=60, bd=5,\
            relief='sunken', height=40, bg="blue")
        self.main_win.Frame2.grid(row = 2, column = 0)

        self.main_win.Frame3 = tk.Frame(self.main_win, width=70, bd=5,\
            relief='ridge', height=40, bg="green")
        self.main_win.Frame3.grid(row = 1, column = 2)

        # Enter the tkinter main loop
        tk.mainloop()

# create an instance of the class
demoInstance = FrameGUI()
```

Running Ex. 5.10 produces this window.



Frame Example

The *LabelFrame* is another container that can be used to arrange controls, that has options similar to the *Frame*, and a *text* option for placing a string inside.

# Chapter 6

## File Handling

File handling in Python uses a *Mode* designation for files when reading and writing. The Python open function creates a file object and associates it with a file in the specified mode. The modes are 'r' for reading (read only), 'w' for writing (contents erased or file created), and 'a' for appending (append to contents or create the file if it does not exist and then append).

Open the file data.txt in read mode. If it does not exist, the statement fails.

```
my_text_file = open('data.txt', 'r')
```

Open the file data.txt in write mode. If it exists, delete the contents. If it does not exist, create the file.

```
my_text_file = open('data.txt', 'w')
```

Open the file data.txt in append mode. If it exists, append to the end. If it does not exist, create the file and append.

```
my_text_file = open('data.txt', 'a')
```

If a path is specified, Python must be told to treat the escape sequence as a literal backslash. Below, the 'r' without quotes before the path accomplishes this.

```
my_text_file = open(r 'C:\Users\Someone\data.txt', 'w')
```

Closing a file is accomplished using the close method and should always be handled deliberately by the program. Ex. 6.1 opens a file, writes some text in the file, closes the file, then re-opens the file for reading, and again closes the file.

## Writing and Reading File Content

### Ex. 6.1 – File handling

```

1 # Creates a File, writes to it, and reads
2
3 def main():
4     outfile = open('newFile.txt', 'w')
5     outfile.write('She sells sea shells.\n')
6     outfile.write('by the sea shore.\n')
7     outfile.close()
8
9     infile = open('newFile.txt', 'r')
10    infile_text = infile.read()
11    infile.close()
12
13    print(infile_text)
14
15 main()

```

Line 4 creates a new file for writing called newFile.txt

Line 5 writes a phrase to the file with a line feed at the end

Line 6 writes another phrase to the file with a line feed

Line 7 closes the file

Line 9 re-opens the file in read mode

Line 10 reads the entire contents of the file into a string

Line 11 closes the file

Line 13 prints the contents of the string

Line 15 calls the main function to execute the program

Example Ex. 6.1 reads the entire contents of the file into a string. Often it is preferred to read one line at a time or even one word at a time. Python provides methods for these as well. This code reads a line from the file *infile*.

```
one_line = infile.readline()    # reads a single line
```



Whether the entire contents of a file is read into a string, or it is read one line at a time, the *split* method can be used to separate the words. It uses whitespace by default as the delimiter when splitting, and can accept any separator. The following line would split comma delimited data.

```
line.split(',') # split using a comma as the separator
```

The following example first reads an entire file into a variable, and splits it into words for output. After closing the file, the program reopens the file and reads one line at a time from the file and splits each line into words for output.

**Ex. 6.2** – Using the split method to separate words after reading a line from a file

```

1 # Uses split to separate words read from a file
2
3 def main():
4     infile = open('newFile.txt', 'r')
5     all_text = infile.read()
6     infile.close()
7
8     for word in all_text.split():
9         print(word)
10
11    infile2 = open('newFile.txt', 'r')
12
13    for line in infile2:
14        for word2 in line.split():
15            print(word2)
16
17    infile2.close()
18
19 main()
```

Line 5 reads the entire contents of the file into *all\_text*.

Line 8 splits the text from *all\_text* into words for each to be output by line 9.

Line 13 reads one line of text at a time from the file into *line*, and the nested FOR loop on line 14 splits the lines into individual words for each to be output by line 15.

## Removing Newline Characters

When writing data to a file, a tab '\t' or newline '\n' is often added to separate data or data sets. Python does not automatically remove the newline character

when reading from a file. The method *rstrip* will remove white space (`\n`, `\t`, and space) from the right side of the string. The string modification methods are shown below.

String modification methods:

<code>lower()</code>	returns a lower case copy of the string
<code>lstrip()</code>	returns a copy of the string with leading white space characters removed
<code>lstrip(char)</code>	returns a copy of the string with leading instances of <i>char</i> removed
<code>rstrip()</code>	returns a copy of the string with trailing white space characters removed
<code>rstrip(char)</code>	returns a copy of the string with trailing instances of <i>char</i> removed
<code>strip()</code>	returns a copy of the string with all leading and trailing white space characters removed
<code>strip(char)</code>	returns a copy of the string with all leading and trailing instances of <i>char</i> removed
<code>upper()</code>	returns an upper case copy of the string

### String Modification Methods

## Reading and Writing Numeric Data

Numbers read from a file are read as strings and must be converted to a numeric data type in order to use them as a numeric value. Chapter 2 introduced casting for type conversion and it can be used when reading from a file, but the data must be correct for the cast or an error will occur. This means removing tabs and linefeeds that may be attached to the data before casting. When a delimiter is present, using `read()` would include the delimiter (tabs in this case) in the returned string. Using `readline()` would also include any delimiter within a line including spaces between items. Conversely, when writing numbers to a file, they must be converted to strings before writing.

Example Ex. 6.3 writes integers to a file using *str* for conversion, then writes their sum converted to a string. Note that the integers are added together and then the result is converted to a string for writing. Attempting to write to a file without the string conversion produces a *TypeError*.

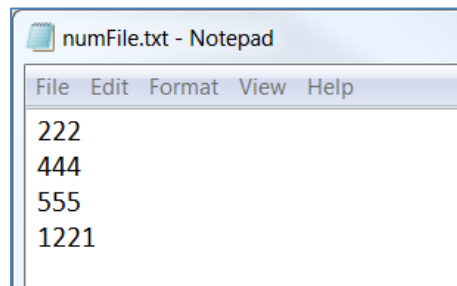
#### Ex. 6.3 – Writing Numbers to a File

```

1 # Program that writes numbers to a file
2
3 def main():
4     outFile = open('numFile.txt', 'w')
5
6     num1 = int(222)
7     num2 = int(444)
8     num3 = int(555)
9
10    outFile.write(str(num1)+'\n'+str(num2)+'\n'+str(num3)+'\n')
11    outFile.write(str(num1 + num2 + num3))
12
13    outFile.close()
14
15 main()

```

Line 10 uses *str* to convert the numbers to strings before writing to the file and adds the newline to each.



Writing Numbers to a File

Reading numeric data requires the same considerations. The elements read from the file must be converted from strings to integers or floats before using them as numeric values.

Example Ex. 6.4 reads three values from the output file of example 6.3 and stores them in three variables. The program then displays the read-in strings and the sum of the values converted to integers, stores the result of the sum, and finally displays it.

## Ex. 6.4 – Reading Numbers from a file

```

1 # Program that reads numbers from a text file
2
3 def main():
4     inFile = open('numFile.txt', 'r')
5     num1 = inFile.readline()
6     num2 = inFile.readline()
7     num3 = inFile.readline()
8     inFile.close()
9
10    print(num1 + num2 + num3)
11    print(int(num1) + int(num2) + int(num3))
12
13    sum = int(num1) + int(num2) + int(num3)
14    print(sum)
15
16 main()
_

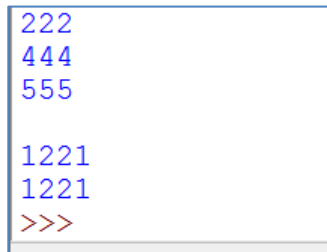
```

Line 10 displays the numbers on individual lines since the line feed was also read. They are treated as strings in the print statement.

Line 11 prints the sum of the individual values converted to integers.

Line 13 adds the values converted to integers and stores the result as an integer in the variable sum.

Line 14 prints the variable sum.



```

222
444
555

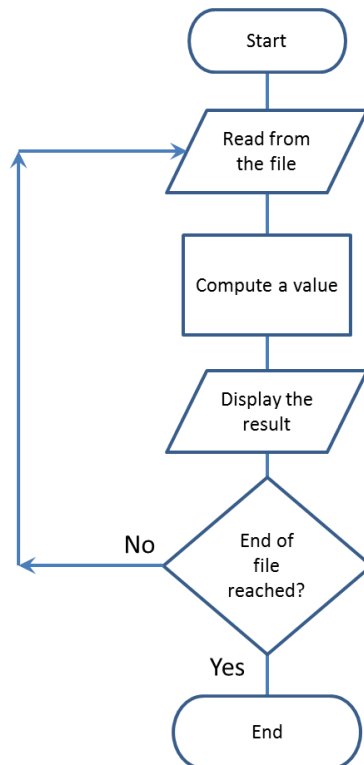
1221
1221
>>>

```

Reading Numbers from a File

The technique used for reading and handling data from a file will be dependent upon the task required. The data can be read one item or line at a time, or the entire contents can be read at once. Loops are typically used when reading one item or line at a time into a variable. The flowchart from Chapter 1 requires a

single computation and display of the result for each item in the file. The code beneath the flowchart is an example of a loop that could be used in this situation using the file from Ex. 6.4.



File Reading Flowchart

**Ex. 6.5** – Reading Numbers from a file and processing them as integers

```

1 # Program that reads/processes numbers from a text file
2
3 def main():
4     inFile = open('numFile.txt', 'r')
5
6     for value in inFile:
7         print(int(value) * 3.5)
8
9     inFile.close()
10
11 main()
  
```

Notice in the output below that the numbers are displayed as floating point numbers. The *promotion* occurs since they are being multiplied by 3.5 (a float).

```
777.0
1554.0
1942.5
4273.5
>>> |
```

Ex. 6.5 Output

The same conversion can be used with floating point numbers. The next example modifies Ex. 6.5 to use *float* instead of integers.

Ex. 6.6 – Reading Numbers from a file and processing them as float

```
1 # Reading numbers from a file and process as float
2
3 def main():
4     inFile = open('floatFile.txt', 'r')
5
6     for value in inFile:
7         print(float(value) * 3.5)
8
9     inFile.close()
10
11 main()
```

```
777.35
1554.7
1943.5499999999997
>>>
```

Ex. 6.6 Output

## Exceptions

Exception handling is required for when a file cannot be created or cannot be opened, and to ensure that there is not a data type mismatch. The format for an exception handler in Python is the *try/except* statement. The general format is:

```
try:
    statement1
    statement2
    etc.

except ExceptionName:
    statement1
    statement2
    etc.
```

The *try* block is entered and if a statement raises an exception, the handler immediately following the *except* clause that matches the type of exception thrown executes and the program continues. An exception that is not handled will halt execution of the program.

#### Ex. 6.7 – Exception Handling

```
def main():
    try:
        inFile = open('missingFile.txt', 'r')

    except IOError:
        print('No File exists.')

    inFile.close()

main()
```

Each type of exception that could be thrown from the try suite should have an exception handler. An exception clause that does not list a specific exception, will handle any exception that is raised in the try suite. This should be the last exception in the series and could be considered a default handler as shown below.

#### Ex. 6.8 – Exception Handling

```
try:
    input_file = open('missingFile.txt', 'r')
    for line in input_file:
        val = int(line)
        sum = sum + val

except IOError:
    print('No File exists.')           # if file cannot be opened

except ValueError:
    print('A bad value was read')     # if data type is incorrect

except:
    print('Error in program.')        # default handler
```

There is a *finally* clause available in Python that executes regardless of whether an exception was raised or not. The finally clause can be used for clean-up like closing a file. Recall that some of the statements in the try block may execute before a statement that throws the exception.

### Ex. 6.9 – Exception handling with finally clause

```

try:
    input_file = open('missingFile.txt', 'r')
    for line in input_file:
        val = int(line)
        sum = sum + val

except IOError:
    print('No File exists.')          # if the file cannot be opened

except ValueError:
    print('A bad value was read')     # if the data type is incorrect

except:
    print('Error in program.')        # default handler

finally:
    input_file.close()                # always executes

```

The try/except statement can also include an *else* clause that will execute only if no exceptions were raised.

Since an exception is an object that contains error information, the exception object can be assigned to a variable and passed to the print function to provide information as shown here.

```

except IOError:
    error = IOError
    print('The error is', error)

```

```

| The error is <class 'OSError'>
| >>> |

```

File dialogs for opening and saving files are covered in Chapter 16.

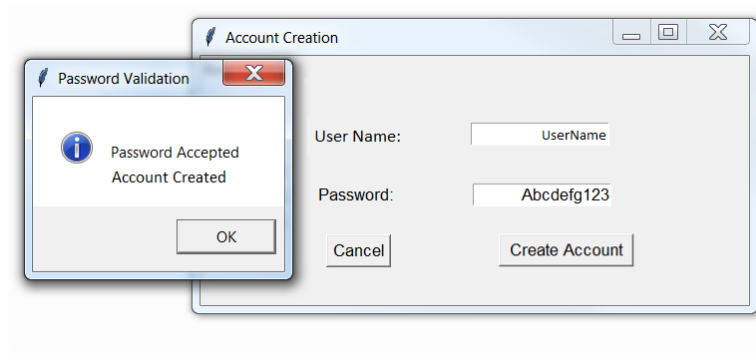


## Chapter 7

# Multiple Windows & Design

### Back to the Project

The project requirements for creating a user account and login will require file handling. When a user logs in, the login information must be verified against current accounts. If the user is creating a new account, it should only be completed if the account does not already exist. Again, checking the current account information is necessary. Both of these scenarios require a file or files for storing and comparing account information. They also require another window for obtaining user input and error handling. Tackling these requirements all at once would be complex, but they can be divided into segments and completed more easily.



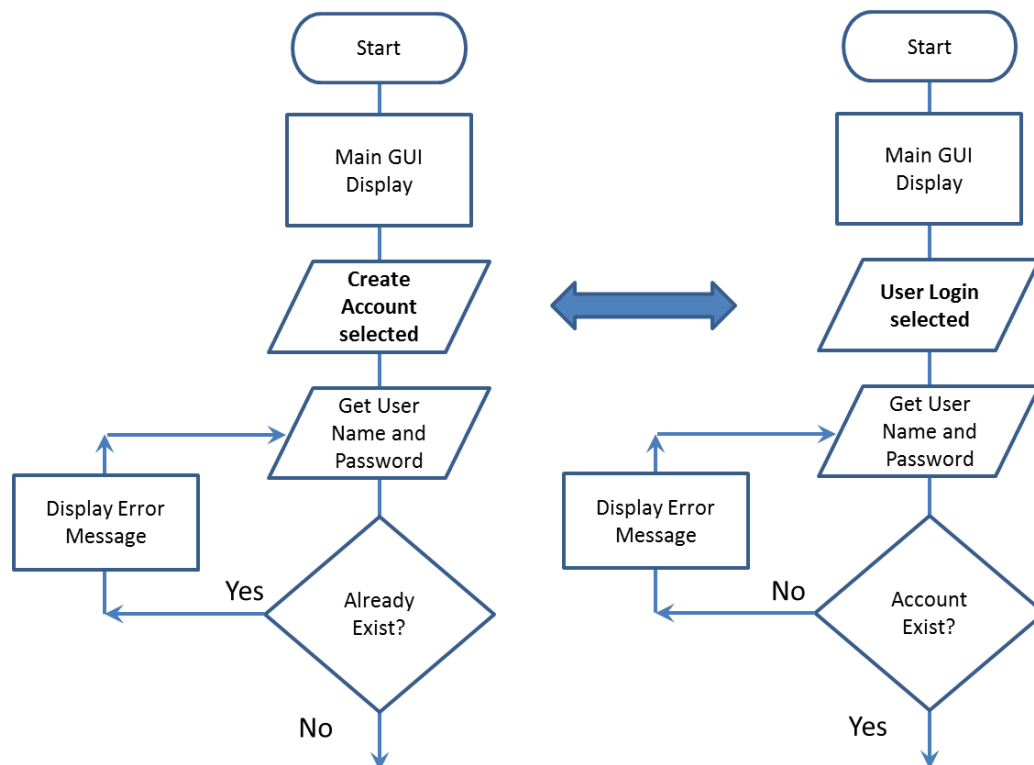
Account Creation Window and Dialog

## Step-wise Refinement and Iterative Enhancement

Breaking down a large problem or task into smaller segments is referred to as **Step-wise Refinement**. A large problem or task is decomposed into smaller tasks and those tasks are then decomposed into even smaller tasks. Once task size and complexity are divided and refined into more manageable pieces, design and development begins.

Once the design stage is completed, the segments are developed and the program is built up as the various parts are completed and added. This process of building and adding software in small segments is referred to as **Iterative Enhancement** and aligns with the *Agile Software Development Process*.

With respect to the User Login and Account Creation operations, there are several areas that can be refined, and a flowchart of the operation shows that there is some commonality.



Account Creation and User Login Flowcharts

There are multiple ways of implementing these operations. One way is to modify the initial window by destroying the controls on it and adding others to accommodate account creation and login. Another way is to use separate windows for each operation and to modularize the program by using multiple classes, files, and functions. This option aligns more closely with Stepwise Refinement, Iterative Enhancement, and an Object Oriented approach.

## Button Click Action

When the Create Account button on the main GUI is clicked, the user will need to enter a username and password. The button click should launch a window with entry components to obtain the information. A command is used to call a function that creates the account creation window. The code for the second window is similar to that of the main GUI with some added features, but it will be created as the result of a button click.

**Ex. 7.1** – Adding a *command* to the Create Account button.

```
self.create_acct_button = tk.Button(text=' Create Account ', width = 16, \
                                   font=("Helvetica",10), command=self.create_account)
self.create_acct_button.grid(row=5,column=1)
```

The function for the command *self.create\_account* is added after the call to *tk.mainloop()*. However, the function is part of the GUI class and it is indented aligning with *def \_\_init\_\_(self)*. This provides two advantages. First, it places the code within the class for access and it allows modification to controls on the main GUI like disabling the Login button. To assist with testing functionality, the function can initially print something as shown here.

```
# Enter the tkinter main loop
tk.mainloop()

# Create the password creation window
def create_account(self):
    print('In the create_account function.') # TEST
```

Testing each time new functionality is added saves debugging time and ensures that each part is working as expected before continuing. As shown, testing the command option for the Login button can be handled with a print statement.

This interim step in the development process is referred as a *stub* which is a function that is incomplete but allows the program to run. It could be a print statement as in the above example, or it could be a function that returns a value indicating that it is incomplete. This is shown in the next example.

Stub example

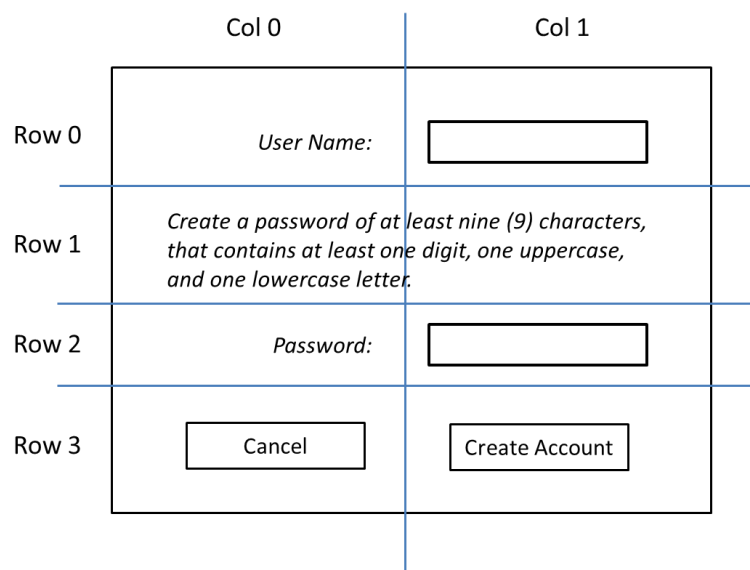
```
def return_user_name():
    return notaname          # return value indicates that it is incomplete
```

The actual call to a function could also be commented, *stubbed out*, to keep it from being called at all.

### Create Account Window Design

The create account window should be designed using the same approach as the main GUI. A sketch using a grid pattern helps to locate the controls on the grid and allows configuring rows and columns prior to writing code. Using the column and row configure methods provides any height or width tailoring required to accommodate positioning.

Ex. 7.2 – Create Account window design sketch.



Account Creation GUI Sketch

## A Second Python Development File (Modularization)

Following a modularization approach, the create account window class code would be in a second Python file and imported into the main Python program file. To create another file for a program, select *File* and then *New File* from the menu on the main program window. An untitled window will open that can be named when it is saved. The name of the file should reflect the code that is in the file. As this program becomes more complex and grows in size, keeping things organized will save time and effort. Some abbreviation or truncation can lessen the amount of typing as long as it doesn't add ambiguity. The goal is to keep it as simple as possible. For the example, the file has been named *create\_acct\_gui.py*. The lines below show the import statement added to the top of the example's main file. Note that the file extension (*py*) for the development file is not used.

```
from tkinter import *           # imports everything from the tkinter module
import tkinter as tk           # imports tkinter as tk

import create_acct_gui         # imports the create_account_gui.py file
```

## Building the Second Window

The code for the Create Account window will be similar in many ways to the main GUI as far as control creation and positioning, but will not have a tkinter loop. The creation of the instance of the AccountGUI will be in the function *create\_account* in the main file.

**Ex. 7.3** – Create Account GUI file – *create\_acct\_gui.py*

1. **# Code for the Create Account window in file create\_account\_gui.py.**
2. `from tkinter import *`
3. `import tkinter as tk`
- 4.
5. `Class AccountGUI:`
6.     `def __init__(acct):`
7.         `print('In the init for AccountGUI')`

For testing purposes, the file has an initialization method and a print function. Notice that the object itself is passed to the *init* method into *acct*. In the assignment statement (line 4 below), *CreateAcctWin* replaces *acct* and the objects data is bound to the object (explained in Chapter 14).

To test the code completed so far, the function in the main file is modified to create an instance of the second window. Recall that this function is below the main loop, but inside the class.

**Ex. 7.4** – Create an Instance of the Create Account Window from main

1. **# Code for the Create Account window.**
2. `def create_account(self):`
3. `print('In the create_account function.')                   # Test`
4. `CreateAcctWin = create_acct_gui.AccountGUI           # instantiation`

As shown on line 4, the file name followed by the class name is used to create an instance of the object which is assigned to *CreateAcctWin*. In the *creat\_acct\_gui.py* file, a print function to test was added.

```
class AccountGUI:
    def __init__(acct):
        print('In the init for AccountGUI')                   # Test
```

The create account window will have three labels, two entry controls, and two buttons as shown in the sketch. To further test and create a window, a name for the window is selected and used to add items to the window. In Ex. 7.5, *acct\_win* is the name of the window object. The parent or master window is *self* (the main GUI). Python programmers often use *root* or *master* in place of *self*.

**Ex. 7.5** – Create Account GUI Window.

```
class AccountGUI:
    def __init__(acct):
        print('In the init for AccountGUI')
        acct.acct_win = tk.Tk()
        acct.acct_win.title("Account Creation")
        acct.acct_win.minsize(width=500,height=250) # set window size
```

Code similar to the main GUI is added including the row and column configure settings, the labels and buttons, and the text *entry controls*.

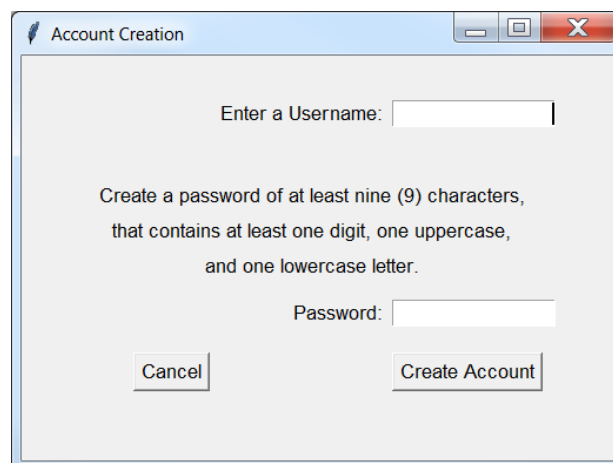
Text entry controls provide a way to position a text box on the window to obtain user input. The designation *acct.acct\_win*, in the code, places the control on the correct window. Self was received as *acct* (which is the master), and *acct\_win* is the actual second window. The options for the entry control include setting a width, justification of the cursor in the entry control using *justify*, and font settings as shown here. To hide input, the code uses the *show* option.

```
acct.acct_win.password_entry = tk.Entry(self.acct_win, width = 15, \
                                       justify='right', font=("Helvetica",10), show='*')
acct.acct_win.password_entry.grid(row=6, column=3, sticky=W)
```

When the window is created, the cursor can be placed in the user name entry control by forcing the focus.

```
acct.acct_win.userName_entry.focus_force()
```

#### Ex. 7.5 – The Create Account Window

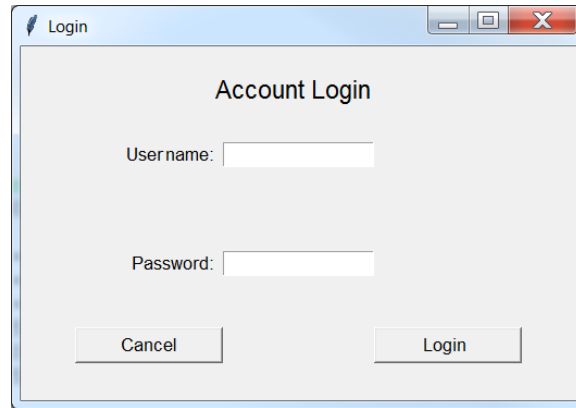


Account Creation Window

In the example, each line of the password requirements is set in a separate row. The button width options are not set, so the buttons are different sizes. The word “Cancel” has fewer letters than “Create Account” so the buttons would not be the same width unless the width is specified as in a previous example.

The login window code will be similar to the account creation code both in main for handling the button click and in the file to create the GUI.

Ex. 7.6 – The Login Window.



User Login Window

## Dialog and Information Boxes

When the user tries to create an invalid password or tries to login using an invalid username or password, an error must be displayed. Very often this is handled using a message box that explains the error and has an “OK” button that must be clicked to continue. The [\*tkinter.messagebox\*](#) module provides information boxes for this purpose and requires importing that module.

Message Box Functions

<code>showerror()</code>	<code>askquestion()</code>	<code>askretrycancel()</code>
<code>showwarning()</code>	<code>askokcancel()</code>	
<code>showinfo()</code>	<code>askyesno()</code>	

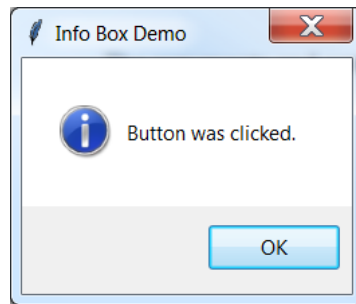
The arguments and options for the functions include:

function name	choice of message box type ( <i>showinfo</i> )
title	the title on the title bar
message	the text to be displayed

The different functions produce different message boxes. The code below produces the standard message box which includes the “OK” button.

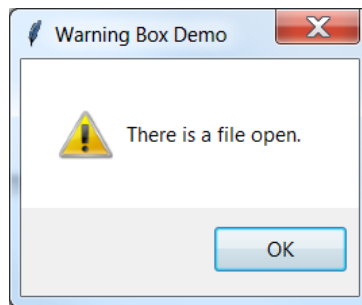


```
tk.messagebox.showinfo('Info Box Demo', 'Button was clicked.')
```



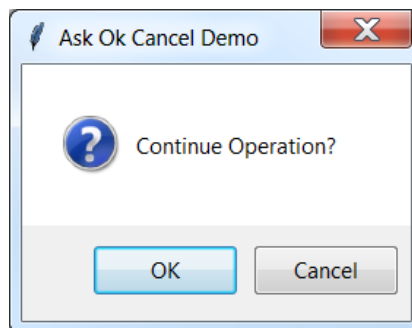
This code produces the warning box shown below.

```
tk.messagebox.showwarning('Warning Box Demo', 'There is a file open.')
```



This code produces the ask-ok-cancel box shown below.

```
tk.messagebox.askokcancel('Ask, OK, Cancel', 'Continue Operation?')
```



Note that the message boxes set the focus for the "OK" button allowing the enter key to be pressed to continue.

Message boxes can provide a way of handling errors and validating input.

## Centering Windows on the Desktop

When windows are created they are displayed in the far top-left corner of the monitor. To center them when they are initially displayed requires getting some display attributes. Obtain the display's screen width and height and subtract the window sizes and divide by two. Then pass the values to geometry again as a string using the format for geometry.

```
self.main.minsize(width = 500, height = 300)    # Establish the window size

# Use screen width and screen height to calculate centering
x_Left = int((self.main.wininfo_screenwidth() - 500)/2)
y_Top = int((self.main.wininfo_screenheight() - 300)/2)

self.main_win.geometry("%dx%d+%d+%d" %(500, 300, x_Left, y_Top))
```

Note the "x" after the first "d" in the statement above. This essentially says that the window should be 500 "by" 300 pixels.

## Bringing a Window to the Front

To raise the window in front of others the lift() method can be used, or attributes('topmost', True)

## Ending a Program and Closing Windows

To determine that the user has ended the program by clicking the "X" in the window, use protocol which can call a function.

```
self.main_win.protocol("WM_DELETE_WINDOW", self.close_prog)
self.sec_win.protocol("WM_DELETE_WINDOW", self.close_prog)

def close_prog(self):
    self.main_win.destroy()
    self.sec_win.destroy()
```

To ensure that a program ends, import sys and use sys.exit().

## Chapter 8

# Data File Design & Data Handling

For the project, storing the user names and passwords and retrieving them for validation during login requires file and data handling. When designing data storage and access, there are a few design and development considerations including the format, text/binary, delimiters, and any encryption. These items must be well thought out during the design phase due to the effect on development of the current program and potential future expansion (scalability). Many large-scale, data intensive programs require a formal *Data Dictionary* which is a separate file that contains the data descriptions, format, delimiters (data separators), the ordering of the data, and often, additional information and comments.

Designing the data format is an important role that has a direct effect on program design and operation, data handling, and the scalability of the data and the program. The data dictionary provides useful information about the file contents and how to extract or parse the data for use in display and analysis.

Creating a data dictionary also allows the file to contain only data and flexibility with respect to delimiters. Data dictionaries are often used to describe the contents of databases and the relationship between its elements.

The sample file data dictionary below specifies individual column numbers for the data elements.

## Data Dictionary Sample from NOAA

DD/MM/YYYY

## GENERAL DATA FORMAT

ONE HEADER RECORD FOLLOWED BY DATA RECORDS:

## COLUMN DATA DESCRIPTION

01-05 STATION NUMBER  
 08-12 RECORDING ENTITY NUMBER  
 14-25 YEAR-MONTH-DAY-HOUR-MINUTE (GMT)  
 27-29 ENTITY DATA RECORD A  
 31-39 ENTITY DATA RECORD B  
 41-45 ENTITY DATA RECORD C  
 47-51 ENTITY DATA RECORD D  
 53-67 ENTITY DATA RECORD E

## Data File Sample

```

USAF WBAN YR--MODAHRMN DIR SPD GUS CLG SKC L M H VSB MW MW MW
724074 93780 200601010054 990 6 *** 4 OVC * * * 5.0 * * * * * * * * 10 * * * *
724074 93780 200601010154 990 3 *** 2 OVC * * * 3.0 * * * * * * * * 10 * * * *
724074 93780 200601010254 300 5 *** 4 OVC * * * 4.0 * * * * * * * * 10 * * * *
724074 93780 200601010354 *** 0 *** 4 OVC * * * 5.0 * * * * * * * * 10 * * * *
724074 93780 200601010406 *** 0 *** 80 OVC * * * 5.0 * * * * * * * * 10 * * * *
724074 93780 200601010454 *** 0 *** 85 OVC * * * 5.0 * * * * * * * * 10 * * * *
724074 93780 200601010459 *** * * * * * * * * * * * * * * * * * * * * * * * * * *
724074 93780 200601010554 *** 0 *** 48 OVC * * * 4.0 * * * * * * * * 10 * * * *
724074 93780 200601010654 250 5 *** 80 OVC * * * 4.0 * * * * * * * * 10 * * * *
724074 93780 200601010754 240 5 *** 75 OVC * * * 4.0 * * * * * * * * 10 * * * *
724074 93780 200601010845 220 3 *** 722 SCT * * * 2.0 * * * * * * * * 10 * * * *
724074 93780 200601010854 230 5 *** 60 BKN * * * 2.0 * * * * * * * * 10 * * * *
724074 93780 200601010908 240 6 *** 60 OVC * * * 4.0 * * * * * * * * 10 * * * *
724074 93780 200601010954 *** 0 *** 50 OVC * * * 5.0 * * * * * * * * 10 * * * *
724074 93780 200601011054 270 3 *** 55 OVC * * * 8.0 * * * * * * * * * * * *

```

Using data such as this is made possible by examining the data while referring to the data dictionary. Although the data file contains a header row, an explanation for the columns is still needed. For instance, the three columns on the right of the sample data are described in the file header row as MW.

The data dictionary indicates that columns 14 thru 25 provide the data and time of the data reading.

14-25 YEAR-MONTH-DAY-HOUR-MINUTE (GMT)

The final row in the sample file for those columns is shown below.

200601011054 (2006 01 01 1054)

The data dictionary makes it clear that this group can be parsed as:

Year 2006, January, 01, and 10:54 Greenwich Mean Time.

For the project user name and password data, there are many possible solutions for storage and access. Both items could be written to a text file on one line with a space or tab between them (columnar data), two lines could be used, or even two files adding a security feature of not having them located together. A binary format could be used instead of text, and encryption could be used as well. Regardless of the storage/retrieval algorithm, the operations are the same with some design choices.

## Create Account Operations

During the Create Account operation, the design could require that the user name be unique, that the password be unique, that each be unique, or that the pair together be unique. In the Login operation, both must be validated as a pair. Considering how the data will be used during login provides insight into how it should be handled in the account creation operation. The functionality should drive the design for storage of the data. Comparing the processes that will utilize the data shows the similarities and the differences for design consideration.

<u>Create Account Operation</u>	<u>Login Operation</u>
1. Get user name	Get user name
2. Get password	Get password
3. Verify as <b>unique</b>	Verify as <b>existing pair</b>
4. Reject errors, go to Step 1	Reject errors, go to Step 1

The only difference in operation is the verification process on Step 3.

Chapter 6 covered file handling from a read and write perspective. The algorithm for account creation and login requires some string manipulation covered in the next chapter in addition to file handling.

## Chapter 9

# Strings, Lists and Tuples

Python provides many ways for string examination and manipulation. The individual characters can be accessed with a FOR loop as shown in chapter 4 (repeated here – temp is a temporary variable that receives a copy of each letter).

```
for temp in "something":
    print (temp)                # displays each letter vertically
```

Strings are sequential and the characters can be accessed by index using square brackets. Index numbering begins at zero, and ends at n-1.

### Ex. 9.1 – Indexing Strings

```
a_string = 'something'
print (a_string[0], a_string[3], a_string[7])    # displays s e n
```

**Negative indexes** can be used to access character positions relative to the last character in the string. The index -1 is the last character in the string.

### Ex. 9.2 – Negative String Indexes

```
b_string = 'negative'
print (b_string[-1], b_string[-4], b_string[-6])    # displays e t g
```

The index can also be used to obtain a copy of a single character from a string.

**Ex. 9.3** – Copying a Character from a String.

```
c_string = 'copy'
ch = c_string[3]           # obtains a copy of y
print(ch)                 # displays y
```

If an out of range index is used, an `IndexError` exception is thrown.

Strings in Python are *immutable*, meaning they cannot be changed once created. The '+' operator is used to concatenate strings. Concatenating a string creates a new string and assigns it to the variable storing the original string. The original string can no longer be used because there is no longer a variable referencing it. Eventually, the Python interpreter will remove the original string from memory. The example below shows the '+' operator used to concatenate strings.

**Ex. 9.4** – Concatenating Strings.

```
d_string = 'New'
d_string = d_string + ' York'
print(d_string)           # displays New York
d_string = 'New'
e_string = ' York'
d_string = d_string + e_string
print(d_string)           # displays New York
```

The `len` function returns the length of a string and can be used as a loop termination condition.

**Ex. 9.5** – The len function with Strings.

```
f_string = 'Length'
print(len(f_string))     # displays 6
index = 0
while index < len(f_string):
    print(f_string[index])
    index += 1           # displays 'Length' vertically
```



## String Slicing and Split

String *slicing* is used to select a portion of a string using a start, end, and step specifier. The general format allows one, two, or three specifiers. When the first specifier is omitted, Python uses zero as the start and the specifier as the end. When two specifiers are used, the first is the start index and the second specifier indexes the end of the slice and is not included in the slice. When three specifiers are used, the third is the step in the sequence.

**Ex. 9.6** – Slice expressions with Strings.

```
sequence = '123456789'
first_four = sequence[:4]
print(first_four)                # displays 1234
second_four = sequence[5:9]
print(second_four)              # displays 6789
every_other = sequence[0:9:2]
print(every_other)             # displays 13579
```

Searching for content in strings is handled using the *IN* and *NOT IN* operators. The following example would find the character '6' in the string and display 'Found a six'.

```
sequence = '123456789'
if '6' in sequence:
    print('Found a six')
```

The *Split* method by default uses the space as a separator and returns a **list** (discussed later in this chapter) of items in the string that are separated by spaces. A different separator can be specified including '/' when a date is being parsed. The split method is accessed using the dot operator.

**Ex. 9.7** – Split a String without a specified separator

```
my_string = 'hour minute second'
time_list = my_string.split()    # split at spaces (default)
print(time_list)                # displays ['hour', 'minute', 'second']
print(time_list[0])             # displays hour
```

Ex. 9.8 – Split a String with a specified separator

```
my_string2 = '10:23:59'
time_list2 = my_string2.split(':')    # split at colons
print(time_list2)                    # displays ['10', '23', '59']
print(time_list2[0])                 # displays 10
```

## String Testing and Modification

The **string testing methods** return true or false and include: `isalnum()`, `isalpha()`, `isdigit()`, `islower()`, `isupper()`, and others.

The **string modification methods** include conversion to upper and lower case, and various strip methods: `lower()`, `upper()`, `lstrip()`, `rstrip()`, and `strip(char)`.

The **search and replace methods** include: `endswith(substring)`, `find(substring)`, `replace(old, new)`, and `startswith(substring)`.

## Lists and Tuples

Lists in Python are sequences of data that are mutable, dynamic, and can be indexed and sliced. They can also hold different types of data. There are no arrays in Python, but lists provide similar functionality. Lists can be iterated over and accessed in the same way as strings.

Ex. 9.9 – Numeric Lists

```
num_list = [5, 15, 25, 35]
print(num_list)                # displays [5, 15, 25, 35]

for n in num_list:
    print(n)                    # displays 5 15 25 35 vertically*

print(num_list[2])             # displays 25
```

\* The numbers are displayed vertically since the print function adds a line feed.

## Ex. 9.10 – String Lists

```

word_list = ['one', 'two', 'three', 'four']
print(word_list)                # displays ['one', 'two', 'three', 'four']

i = 0
while i < len(word_list):
    print(word_list[i])         # displays one through four vertically
    i = i + 1                  # could be written as i += 1

print (word_list[2])           # displays three

```

Note the difference in the displayed output between the print function for the entire list and the output from the loops. The brackets are removed since the loop iterates over the list and copies the element in the list to the variable.

List elements can be changed and lists can be concatenated using the '+' operator, and added to or removed from. The len function works with them as well.

```

first_list = ['a', 'b', 'c', 'd']
print(first_list)               # displays ['a', 'b', 'c', 'd']

first_list[1] = 'z'
print(first_list)              # displays ['a', 'z', 'c', 'd']

second_list = ['e', 'f', 'g', 'h']
first_list = first_list + second_list # concatenated lists
print(len(word_list1))         # displays 8
print(first_list)              # displays ['a', 'z', 'c', 'd', 'e', 'f', 'g', 'h']

```

There are also **built in functions** for lists including: `append()`, `insert(index, item)`, `sort()`, `remove(item)`, `reverse()`, `min(list_name)`, and `max(list_name)`.

Lists can be passed to functions and functions can return lists. In the next example, the list `num_list` is passed to `get_sum` which returns a sum of the numbers in the list.

**Ex. 9.11** – Lists as arguments to functions

```
def main():
    num_list = [5, 15, 25, 35]
    print('The sum of the list is :', get_sum(num_list))

def get_sum(in_list):
    sum = 0
    for num in in_list:
        sum = sum + num

    return sum

main()
```

Lists can be written to files with *writelines(list\_name)*, but there are no line feeds with this method. To include line feeds, a loop is needed and the newline character needs to be added.

```
for item in my_list:
    outfile.write(item + '\n')
```

A tab or a space could be added the same way and used as a delimiter when reading.

A *tuple* is simply a list that is constant and cannot be changed. Tuples process faster and the fact that the data cannot be changed in a tuple protects the data. Tuples can however be converted to lists, and lists to tuples.

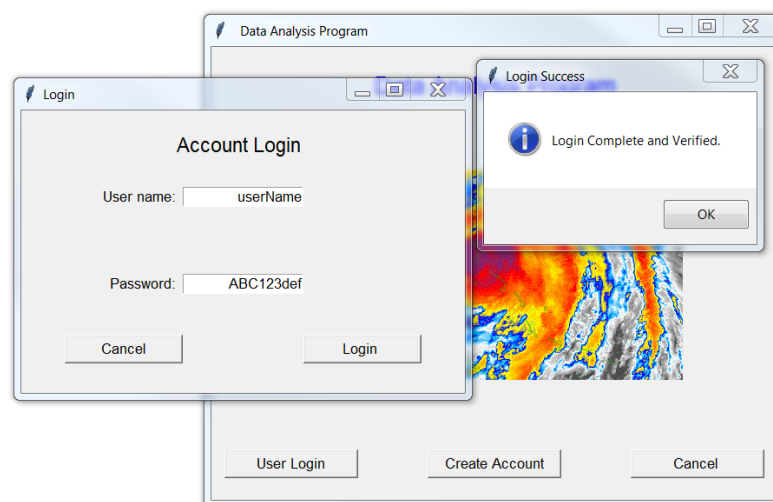
```
my_tuple = tuple(my_list)           # convert list to tuple
my_list2 = list(my_tuple)          # convert tuple to list
```

## Chapter 10

# Remove, Modify, or Hide Controls

### Back to the Project

Once an account has been created and the login successful, rather than creating a new window, the main window can be modified to become the main interface for the program. The login window can be destroyed and control can be returned to the main window, which can then be modified for program interaction. This will require changes to or removal of the existing controls.



Successful Login

The commands to destroy controls differ depending on what is being destroyed, and in some cases reconfiguring or hiding the item is preferred to removal. The main window has a command to handle the situation when the user cancels the program that destroys the main window. Notice that in this case, the destroy function has no parenthesis.

```
self.quit_button = tk.Button(text='Cancel', font=("Helvetica",10), \
                             command=self.main_window.destroy)
```

To destroy the create account and login windows, the function destroy has parenthesis.

```
self.acct_win.destroy()           # destroy the window
```

In those cases when it is preferred, destroy can be used for most controls.

```
self.userName_label.destroy()    # destroy a label
self.user_entry.destroy()        # destroy an entry control
```

The grid geometry manager also has *forget* and *remove* methods. Both of these methods remove the control from the grid manager. The control is not destroyed, and can be redisplayed by grid or any other manager.

```
grid_forget()                    # control is not destroyed
grid_remove()                    # control is not destroyed
```

Controls can also be reconfigured instead of being destroyed using the *config* method. The config method allows modification to controls as long as the grid positioning statement is on a separate line when the control is created.

```
self.heading_label = tk.Label(self.main_window, text="First Text", \
                              font=("Helvetica",16), fg="blue")
self.heading_label.grid(row=1,rowspan=2, column=1)

self.heading_label.config(text='New Text', font=("Arial",16), fg="black")
self.heading_label.grid(row=3,column=3,columnspan=5,rowspan=2)
```

## StringVar

The `tkinter` module provides the `StringVar` class. The `StringVar` modifies any control that uses it whenever the `StringVar` is changed. This provides the ability to have an immediate update to a control anytime the value that is stored in the `StringVar` object changes. A `StringVar` is declared and assigned to a control.

```
my_svar = tk.StringVar()

my_label = tk.Label(textvariable= my_svar)
```

The update to the `StringVar` can come from a variety of sources including button clicks. The `StringVar`'s `set` method is used to change the text. In this example, a simple window (code omitted) is created with: a heading label, `StringVar`, label for the `StringVar`, and a button. The `StringVar` is changed each time the button is clicked using the command `self.change_text`.

**Ex. 10.1** – `StringVar` modified from a button click.

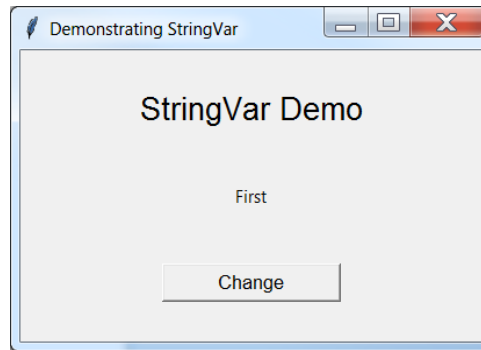
```
self.num = 1                                # variable used by the function

self.heading_label = tk.Label(self.main_window, text='StringVar Demo', \
                               font=("Helvetica",16))
self.heading_label.grid(row=0, rowspan=2, column=1)

# Label with StringVar
self.sVar = tk.StringVar()                  # StringVar
self.sVar.set('First')
self.changing_label = tk.Label(textvariable=self.sVar, font=('Tahoma',10))
self.changing_label.grid(row=2, column=1)

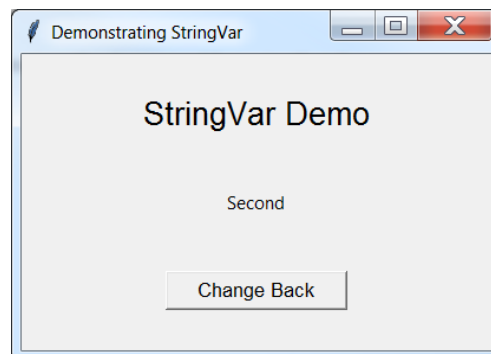
self.change_button = tk.Button(text='Change', \
                                width=16, font=("Helvetica",10), \
                                command=self.change_text)      # command
self.change_button.grid(row=3, column=1)

# Enter the tkinter main loop
tk.mainloop()
```



When the button is clicked, the command calls the function and changes the StringVar using the set method. To toggle the changed text for the example, the variable num is reassigned each time. Note that the button text is also changed each time using config.

```
def change_text(self):  
    if self.num == 1:  
        self.sVar.set('Second')  
        self.change_button.config(text='Change Back')  
        self.num = 2  
    else:  
        self.sVar.set('First')  
        self.change_button.config(text='Change')  
        self.num = 1
```



In addition to the StringVar, destroying, forgetting, and removing controls allows modification to existing GUI controls instead of creating a completely new windows.



# Chapter 11

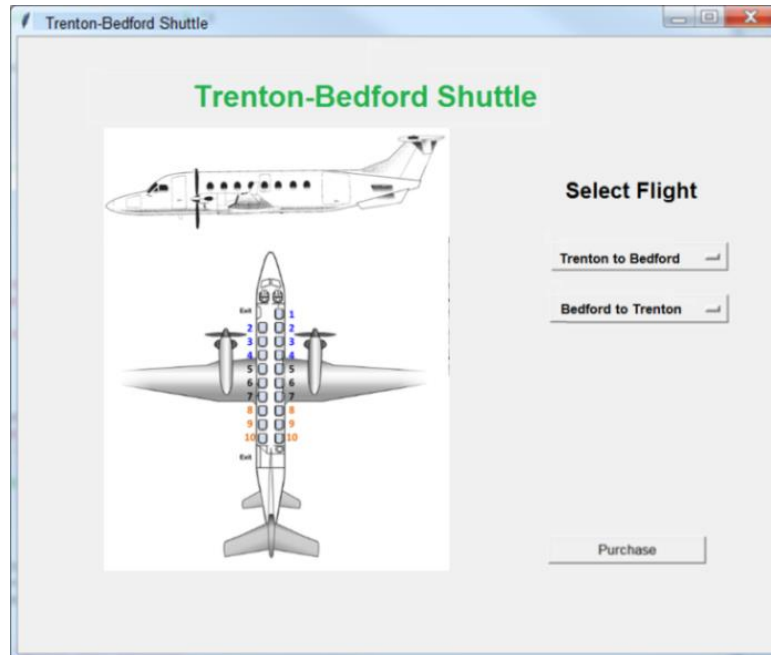
## Main Interface GUI

Design is the first step to developing the main interface. The user will interact with the program through this GUI and ease-of-use, intuitive controls, and descriptive labels are necessary. The controls to be used on the main GUI depend on what the program does and user interaction. A few buttons may be adequate or the program may require a more complex layout. Button clusters can be used to allow multiple selections and radio buttons and drop-down menus are mutually exclusive requiring the user to select just one. These considerations during the design phase will save the time spent redesigning or reconfiguring an inadequate or problem interface.

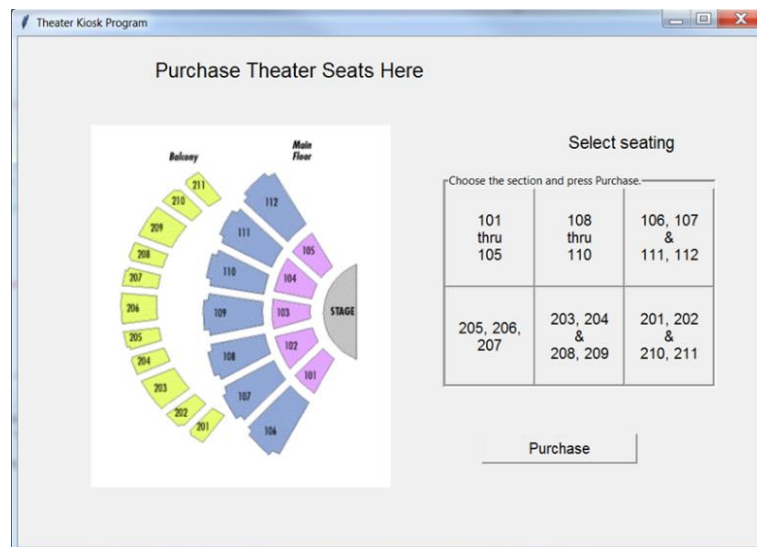
The interface layout should be designed in conjunction with the operational design. Storyboarding, pseudo-code, and flowcharts used during design will show issues that can be corrected early in the process. Software engineers often overlook essential aspects of the interface since they know what the program does, how it functions, and the inputs required. The Agile process typically involves stakeholder reviews and in some cases the client or customer is involved. This provides an opportunity for people not familiar with the planned design and operation of the program to offer suggestions for improvement. It also eliminates surprises when the final product is delivered.

Several examples of interfaces for other projects are shown here.

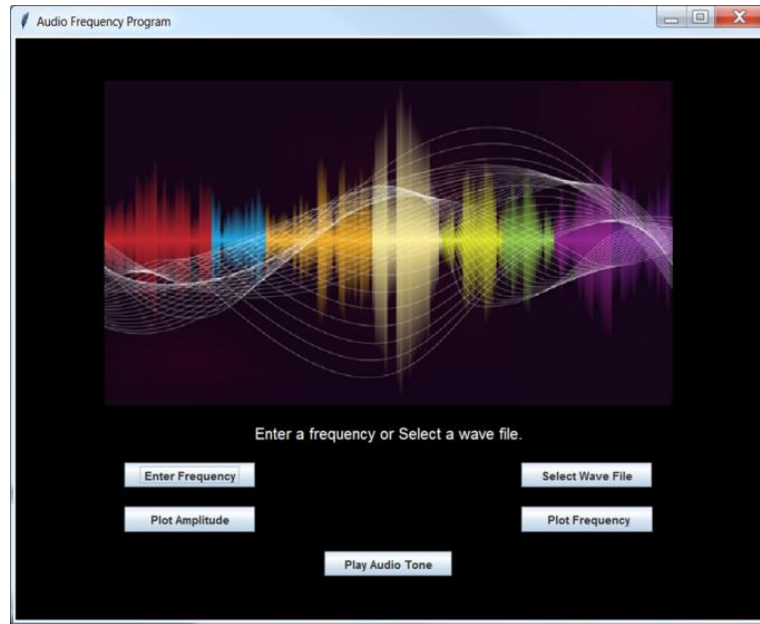
The **Option Lists** in this example provide the selection input and the image corresponds to the option-list choices. The purchase button is not enabled until a seat selection has been made.



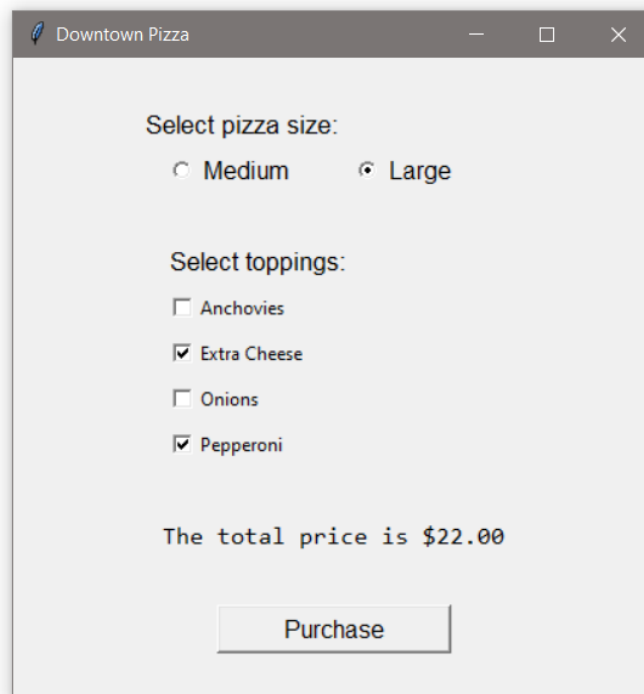
A **button cluster** is shown in this example for the user selection. In addition, there is an image corresponding to the choices, and a purchase button



In the following example, only buttons are used and they are enabled and disabled to prevent errors. For example, a frequency cannot be played or plotted if one has not been entered or selected.



This example uses radio buttons and check boxes for selection.



The same design tools that were used for the initial window can be employed including the row and column sketch for placement of the controls and image(s). The configure option can then be used to set the rows and columns as needed to accommodate the elements of the window.

The next chapter covers control examples including those shown above.

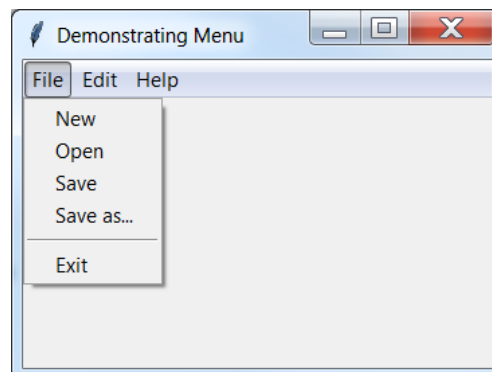
# Chapter 12

## Menus and Button Groups

User selection of operations is often best handled with a control to eliminate typing errors and to reduce input validation requirements. The pull-down menus and button groups shown in chapter 11 provide interaction with the user while preventing errors. The implementation of those elements and placements in the window using a grid are handled much the same way as the Label controls covered in earlier chapters. One exception is the drop-down or window menu.

### Drop-down (window) Menu

A drop-down or window menu can provide program-level operations. The menu rests on the window frame, and drops down to reveal the options.



Drop-down Menu

The Python drop-down is created using `Menu()`. The items listed on the menu are added using `add_command` and a function to respond to the selection. There is a separator that can be added between selections, and “`tearoff`” allows the menu to actually be pulled away from the border (set to zero below which turns off that feature).

**Ex. 12.1** – Drop-down Menu.

```
class DemoGUI:
    def __init__(self):
        self.main_win = tk.Tk()
        self.main_win.title("Demonstrating Menu")
        self.main_win.minsize(width=350,height=180)

        self.menubar = Menu()
        self.filemenu = Menu(self.menubar, tearoff=0) # tearoff option
        self.filemenu.add_command(label="New", command=self.temp)
        self.filemenu.add_command(label="Open", command=self.temp)
        self.filemenu.add_command(label="Save", command=self.temp)
        self.filemenu.add_command(label="Save as...", command=self.temp)
        self.filemenu.add_command(label="Close", command=self.temp)
        self.filemenu.add_separator()
        self.filemenu.add_command(label="Exit", command=self.main_win.destroy)
        self.menubar.add_cascade(label="File", menu=self.filemenu)

        -
        - # Other code here
        -
        -
        self.main_win.config(menu=self.menubar)

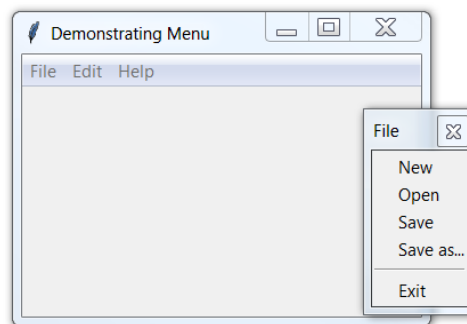
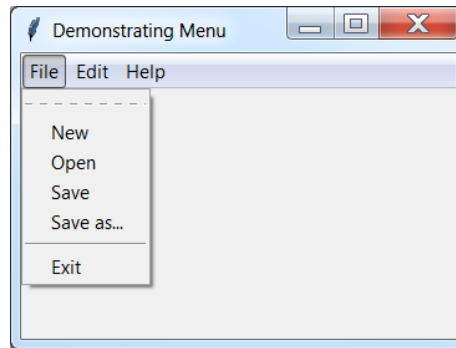
        tk.mainloop()

    def temp(self):
        print("Menu item selected.")

demo = DemoGUI()
```

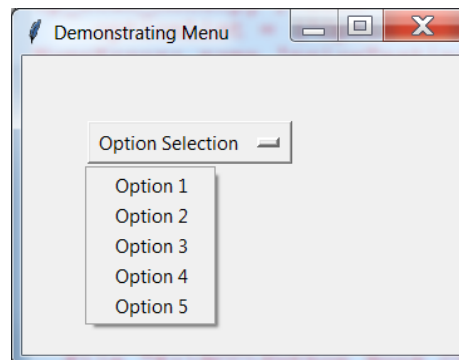
Changing the *tearoff* assignment to 10 instead of zero places a dotted line on the drop-down menu. This can be used to “tear off” the menu by clicking on the dotted line and dragging the menu to another location.

```
self.filemenu = Menu(self.menubar, tearoff=10)
```



## Option List

An option list allows locating a list of selectable options anywhere using grid.



Option List

## Ex. 12.2 – Option List

```

class DemoGUI:
    def __init__(self):
        self.main_win = tk.Tk()
        self.main_win.title("Demonstrating Menu")
        self.main_win.minsize(width=350,height=180)
        self.main_win.rowconfigure(0, minsize = 50)
        self.main_win.rowconfigure(2, minsize = 150)
        self.main_win.columnconfigure(0, minsize=50)

        optionList = ('Option 1', 'Option 2', 'Option 3', 'Option 4', 'Option 5')
        self.option_var = tk.StringVar()
        self.option_var.set('Option Selection')
        self.option_menu = tk.OptionMenu(self.main_win,\
                                         self.option_var, *optionList)
        self.option_menu.grid(row=1, column=1)

        tk.mainloop()

# create an instance of the class
demo = DemoGUI()

```

To obtain the selection from the user, a command is linked to the option list.

```

self.option_menu = tk.OptionMenu(self.main_win, self.option_var, \
                                 *optionList, command = list_changed)

```

The function called will need to receive the arguments and use the *get* method to obtain the selection.

```

# on change drop-down value
def list_changed(self, *args):
    print(self.option_var.get())

```



In the next example, the “room” option list is disabled until a “floor” is selected. The “trace” attribute calls “get\_floor\_info” when the option list changes.

**Ex. 12.2A** – Option Lists – enable/disable call functions

```

floorOptionList = ('Ground Floor', 'Second Floor', 'Third Floor')
self.floorOption_var = tk.StringVar()
self.floorOption_var.set('Select Floor')
self.floorOption_menu = tk.OptionMenu(self.main_win,
                                     self.floorOption_var, *floorOptionList)
self.floorOption_menu.grid(row = 1, column = 1)
self.floorOption_var.trace('w',self.get_floor_info)

roomOptionList = ('King Room ', 'Twin Room ', ' Deluxe King Room ', \
                  'Corner King Room ', 'Corner Suite ')
self.roomOption_var = tk.StringVar()
self.roomOption_var.set('Select Room')
self.roomOption_menu = tk.OptionMenu(self.main_win,
                                     self.roomOption_var, *roomOptionList)
self.roomOption_menu.grid(row = 3, column = 1)
self.roomOption_menu.config(width=18, state='disabled')

```

The “get\_floor\_info” function activates the “room” option list (note the parameters). A print statement will show what they are.

```

def get_floor_info(self, a, b, c):
    print('a is ' +str(a) + ' and b is ' + str(b) + 'and c is ' + str(c))
    floorSelect = self.floorOption_var.get()
    if floorSelect == 'Select Floor':
        tk.messagebox.showinfo('Invalid floor','Please select a floor.')
    else:
        self.roomOption_menu.config(state='active')

```

The “Select” button calls “get\_room\_info”.

```

self.select_button = tk.Button(self.main_win, text = 'Reserve',
                               width=16, font=('Helvetica',14), command=self.get_room_info)
self.select_button.grid(row=5, column=1)

def get_room_info(self):
    roomSelect = self.roomOption_var.get()
    print('The room was ' + roomSelect)

```

## Button Groups and Clusters

Groups of buttons are often preferred to lists. The example below creates a group of six buttons on a label frame that has a sunken border. The buttons are created from a list and are positioned using a loop as opposed to individually placed.

**Ex. 12.3** – Button Group.

```

self.lf = tk.LabelFrame(text = "Choose a button.", padx=6, pady=16,\
                        bd=4, relief= SUNKEN)
self.lf.grid(row= 1, rowspan=4, column = 1, columnspan=4)

btn_list=['B #1', 'B #2', 'B #3', 'B #4', 'B #5', 'B #6']

keyRow = 0
keyCol = 0
index = 0
button_num = ""                                # empty string for button number

btn = list(range(len(btn_list)))

for button_num in btn_list:                    # start of for loop
    cmd = lambda btn_num = button_num: self.button_clicked(btn_num)

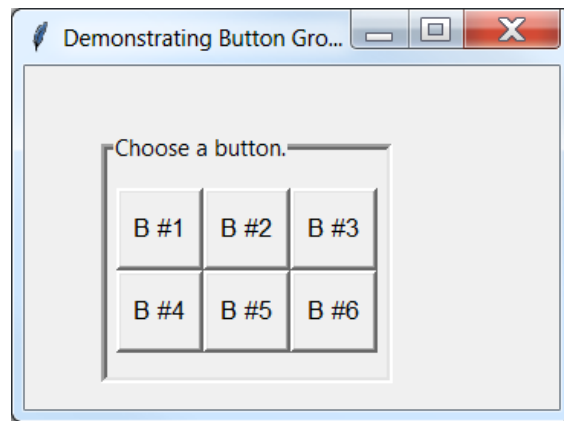
                                                # create the button
    btn[index] = tk.Button(self.lf, text= button_num, font=("Helvetica",10), \
                            height= 2, width=5, bd=3, command=cmd)

                                                # position the buttons
    btn[index].grid(row= keyRow, column= keyCol)

    index = index + 1                          # increment button index and row
    keyCol = keyCol + 1
    if keyCol > 2:
        keyCol = 0
        keyRow = keyRow + 1

```

The label frame used in the example has a `SUNKEN` appearance using the `relief` option. The buttons are arranged using the loop and rows/column variables. The list of buttons is stored in `btn` and the index is used to access the individual buttons.



Button Group

The command for the buttons is assigned `cmd` which is a lambda that calls the `button_clicked` function passing it the text on the button.

```
def button_clicked(self, btn_num):
    print("the key is " + str(btn_num))
```

```
Button_Group.py
the key is B #6
the key is B #5
the key is B #1
>>>
```

## Lambda Expressions

A lambda expression is an inline function with no name. In Python, the keyword `lambda` or `def` can be used along with or without a name. The two lines below accomplish the same thing.

```
def square_root(x): return math.sqrt(x)

square_root = lambda x: math.sqrt(x)
```

Lambda expressions are not necessary, but can make writing the code easier. When a function is simple and is called only once, a lambda expression makes sense. It can be anonymous (no name) and defined where it will execute.

One frequent use of a lambda is in programming “callbacks” (explained later) to GUI frameworks such as Tkinter and wxPython. A situation to use a lambda is the command assigned to a button. A `tk.Button` requires a function object to be assigned to the command. A way of handling this is to have the command be a call to a function and then to have that function perform the operation.

#### Ex. 12.4 – Button Command for Print Function

```
self.new_button1 = tk.Button(text='Button 1', width=16, font=("Arial",10),\
                             command=self.on_click)

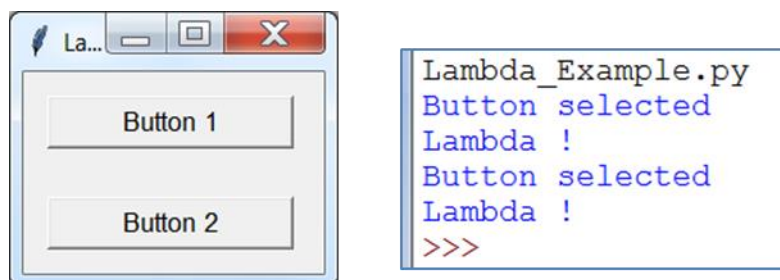
def on_click(self):
    print('Button selected')
```

In the above example, the print command cannot be assigned directly to the button. The command must call the function `on_click` which then handles the print function. Using a lambda function would eliminate the call to the function as shown below.

#### Ex. 12.4A – Lambda Button Command

```
self.new_button2 = tk.Button(text='Button 2', width=16, font=("Arial",10),\
                             command=lambda : print('Lambda !'))
```

Combining the two buttons and the function into a single program shows that both versions operate the same way.



Function/Lambda Example

When a GUI program uses this type of code, the button object is said to “call back” to the function object that was supplied to it as it’s *command*.

## Radio Buttons

Very often the design or operation of the program requires that only one selection be made by the user. Radio buttons are a great solution to this problem. They are *mutually exclusive*, and when a button is checked the button that was previously checked is unchecked. The value assigned to the button can be captured and used. The options for radio buttons are similar to other controls including text and fonts, and they are located using grid locations. The code below uses *radio\_var* to store the value from the radio buttons. Note that both buttons are assigned the same variable *radio\_var*, but are assigned a different integer in *value*. After a selection is made, the user clicks on the *Display Data* button that has a command that calls *rad\_react* which obtains the value of the selected radio button using *get()*. To use them, import from *tkinter* import *\**.

Ex. 12.5 – Radio Buttons (note: portions of the code is omitted)

```

self.radio_var = tk.StringVar()           # to store the radio button selection
self.radio_var.set('1')                  # the default button selected

                                           # barometric pressure radio button
self.baro_press = tk.Radiobutton(text = " Barometric pressure",\
                                font=("Arial",12), variable=self.radio_var, value='1')
self.baro_press.grid(row=3, column=1, sticky=W)

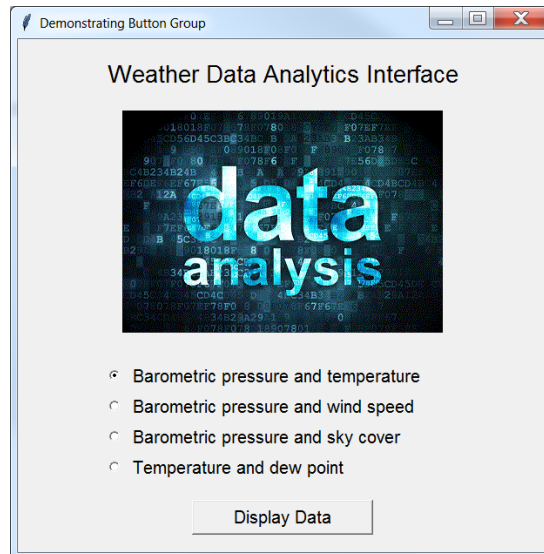
                                           # wind speed radio button
self.wind_spd = tk.Radiobutton(text = " Wind speed", \
                               font=("Arial",12), variable=self.radio_var, value='2')
self.barowind_spd.grid(row=5, column=1, sticky=W)

self.display_data_button = tk.Button(text='Display Data', width=18, \
                                     font=("Helvetica",12), command = self.rad_react)
self.display_data_button.grid(row=11,column=1)

```

The function below is called by the button to obtain the number assigned to the selected radio button.

```
def rad_react(self):                                # function called by the button
    print('Button selected is ' + str(self.radio_var.get()))
```



Ex. 12.5 Radio Button Example

Selecting each of the radio buttons produces the following output with the number of the radio button.

```
Radio_Buttons.py
Button selected is 1
Button selected is 2
Button selected is 3
Button selected is 4
```

Ex. 12.5 Radio Button Example Output

# Chapter 13

## Date and Time

Time values in Python are handled using the *Time* class with attributes for hour, minute, second, and microsecond. There are also multiple optional arguments for an instance of the time object to set a clock in a program. Obtaining the current date and time is straight forward.

### Ex. 13.1 – Time Example

```
print ("Current date and time: " , datetime.date.time.now())
tm = datetime.date.time.now()
print('hour : ', tm.hour)
print('minute:', tm.minute)
print('second:', tm.second)
print('microsecond:', tm.microsecond)
```

The output for Ex. 13.1 on September 26, 2020 at 8:01 PM is:

```
Current date and time: 2020-09-26 20:01:04.665855
2020-09-26 20:01:04.681455
hour : 20
minute: 1
second: 4
microsecond: 681455
```

The date and time information can also be stored in a tuple, which allows extraction of the individual pieces as needed.

#### Ex. 13.2 – Time Tuple Example

```
tm = datetime.datetime.now()
print ("Tuple: " , tm.tuple())
```

The output for Ex. 13.2 on September 26, 2020 at 8:11 PM is:

```
Tuple: time.struct_time(tm_year=2020, tm_mon=9, tm_mday=26,
tm_hour=20, tm_min=11, tm_sec=1, tm_wday=3, tm_yday=269, tm_isdst=-1)
```

Calendar dates are handled using the *date* class with attributes for year, month, and day. To obtain the current date, the *today()* class method is used.

#### Ex. 13.3 – Date Example

```
dm = datetime.date.today()
print(dm)
print('today: ', dm.day)
print('month: ', dm.month)
print('year: ', dm.year)
print ('tuple:', dm.timetuple())
ts = dm.timetuple()
print('The day is: ', ts[2])
```

The output from Ex. 13.3 on September 28, 2020 at 6:30 PM is:

```
Current date and time: 2020-09-28 18:30:22.284823
2020-09-28
today: 28
month: 9
year: 2020
tuple: time.struct_time(tm_year=2020, tm_mon=9, tm_mday=28,
tm_hour=0, tm_min=0, tm_sec=0, tm_wday=5, tm_yday=271, tm_isdst=-1)
The day is: 28
```



## Calendar

Python has a calendar module that provides various calendars and options for displaying and using them. The module includes a TextCalendar for real text and HTML for special formatting.

The following code prints a calendar for the month of September. Notice that the start day must be set in order for the first day to be Sunday. The default is Monday which is a European convention.

**Ex. 13.4** – Display a Calendar of One Month

```
yy = 2021
mm = 7
calendar.setfirstweekday(calendar.SUNDAY)    # all uppercase
print(calendar.month(yy,mm))
```

July 2021						
Su	Mo	Tu	We	Th	Fr	Sa
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31

Ex. 13.4 Month Calendar Output

To display an entire year, the month attribute is omitted.

**Ex. 13.4A** – Display a Calendar for the Year

```
yy = 2021
calendar.setfirstweekday(calendar.SUNDAY)
print('\n\n')
print(calendar.calendar(yy))
```

```

Python 3.9.0 Shell
File Edit Shell Debug Options Window Help

2021

January
Su Mo Tu We Th Fr Sa
    1 2
 3 4 5 6 7 8 9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
31

February
Su Mo Tu We Th Fr Sa
 1 2 3 4 5 6
 7 8 9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28

March
Su Mo Tu We Th Fr Sa
 1 2 3 4 5 6
 7 8 9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29 30 31

April
Su Mo Tu We Th Fr Sa
    1 2 3
 4 5 6 7 8 9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30

May
Su Mo Tu We Th Fr Sa
    1
 2 3 4 5 6 7 8
 9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30 31

June
Su Mo Tu We Th Fr Sa
    1 2 3 4 5
 6 7 8 9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30

July
Su Mo Tu We Th Fr Sa
    1 2 3
 4 5 6 7 8 9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30 31

August
Su Mo Tu We Th Fr Sa
 1 2 3 4 5 6 7
 8 9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30 31

September
Su Mo Tu We Th Fr Sa
    1 2 3 4
 5 6 7 8 9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30

October
Su Mo Tu We Th Fr Sa
    1 2
 3 4 5 6 7 8 9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
31

November
Su Mo Tu We Th Fr Sa
 1 2 3 4 5 6
 7 8 9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29 30

December
Su Mo Tu We Th Fr Sa
    1 2 3 4
 5 6 7 8 9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30 31

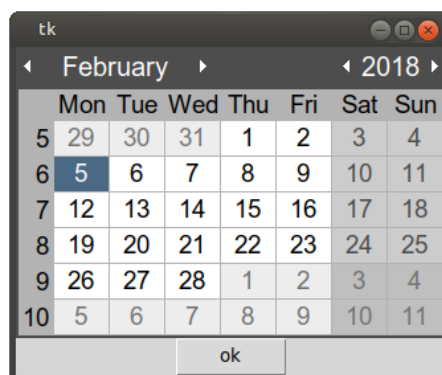
Ln: 18 Col: 0

```

Ex. 13.4A Year Calendar Output

There are many methods associated with calendars including *day\_name*, *day\_abbr*, *month\_name*, and *month\_abbr*.

In addition, `tkcalendar` is a python module that provides the `Calendar` and `DateEntry` controls for Tkinter. The `DateEntry` control is similar to a `Combobox`, but the drop-down is not a list but a `Calendar` to select a date.



tkcalendar Display

# Chapter 14

## Displaying Data

Many GUI programs display data to the user. This can be handled much like file writing in Python. There are two situations to consider; handling output to a display as the user enters data for a computation, and reading data from a file to display.

The examples below use a program that computes a loan payment based on a user's input of the loan amount, interest rate, and duration of the loan. Entry controls on the main window obtain the user input and a button click calls a compute function to compute the monthly payment amount. A *StringVar* is used to update the output label for the monthly payment amount on the main window. The second display of a computation history will be addressed later in the example.

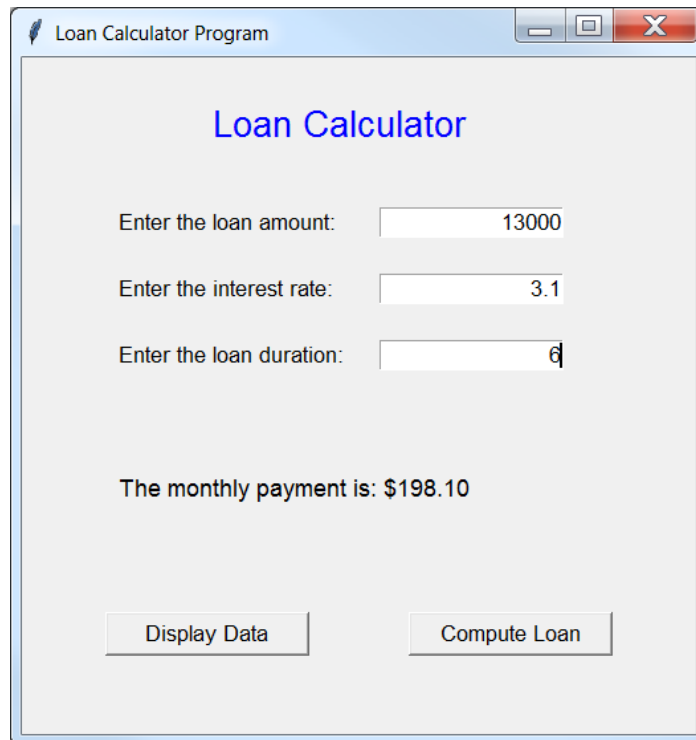
StringVar Code

```
self.pymt_var = tk.StringVar()

self.payment_label=tk.Label(textvariable=self.pymt_var,\
                             font=("Arial",12))
self.payment_label.grid(row=6,column=2,sticky=W)

self.pymt_var.set('The monthly payment is: ')
```

## Ex. 14.1 – Loan Payment Example



Loan Payment Example

The payment amount is displayed on the main window by modifying the `StringVar` in the function. A change to the `StringVar` using `set()` automatically updates the label.

```
# Updating a stringvar
new_str = 'The monthly payment is: $' + str(format(mp, '.2f'))
self.pyamt_var.set(new_str)           # update the stringvar
```

The history of loan computations will be displayed in a second window. This window will not be a user interface, but simply, a display window of the text. The “Display Data” button on the GUI will create the second window.

Before continuing, it is useful to include a further explanation of object creation and the particulars of `self/root/master` (interchangeable names) and what is passed to and received by the `__init__` method. In the `init` method, `self` refers to the new object, but in other class methods, it refers to the object instance of the method called. This will be further explained in the next section.

## Window Interaction

To demonstrate object interaction and what is actually passed into and received by the `__init__` method, the following example creates a main window and then a second window using two classes. The main window will create an instance of the second window and then modify what is displayed in the second window.

### Ex. 14.2 – Main Window

```

from tkinter import *      # imports everything from the tkinter module
import tkinter as tk      # imports tkinter as tk

class MainGUI:
    def __init__(self):
        self.main_win = tk.Tk()          # create the main window
        self.main_win.title("The First Window")
        self.main_win.minsize(width=400,height=200) # window size

        self.heading_label = tk.Label(self.main_win, text='First Win', \
                                       font=("Helvetica",16), fg="blue")
        self.heading_label.grid(row=1,rowspan=2,\
                                column=3, columnspan=2)

        self.print_button = tk.Button(text='Print', width=12,\
                                       font=("Helvetica",10), command=self.output)
        self.print_button.grid(row=3,column=1, columnspan=2)
        print('In main and self is: ', str(self))
        tk.mainloop()                  # enter the tkinter main loop

    def output(self):
        print ('Self is now: ', str(self))

FirstWin = MainGUI()                  # create an instance of the class

```

In the code in Ex. 14.2, an instance of the class `MainGUI` is created as “FirstWin” by the assignment statement (last line of the program) `FirstWin = MainGUI()`.

The window has a heading label and button labeled “Print” that will be used to call the function below the main loop `def output(self)`. There is also a print statement above the main loop. Notice first that the command does not pass any arguments, yet the output function receives one. This is the implicit passing of `self` in Python.

When the program runs, the output of `self` in the initialization and the output produced by clicking the “Print” button are the same hexadecimal memory address.

```
In main and self is: <__main__.MainGUI object at 0x03BE2330>
Self is now: <__main__.MainGUI object at 0x03BE2330>
```

#### Ex. 14.2 Output

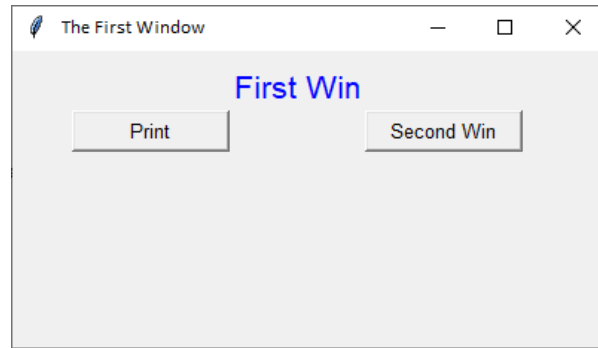
The second window will be created using a second button on the main GUI, and the command for the button will call the function `second_win` as shown below. A few additions including configure loops as shown here, were made to the program for convenience.

**Ex. 14.3** – Additions to the `MainGUI` class to add a second button and function

```
for r in range(6):
    # configure loops
    self.main_win.rowconfigure(r, minsize=10)
for c in range(6):
    self.main_win.columnconfigure(c, minsize=40)

self.second_win_button = tk.Button(text='Second Win', width=12,\
                                   font=("Helvetica",10), command=self.second_win)
self.second_win_button.grid(row=3,column=5, columnspan=2)

# New function added
def second_win(self):
    print('In second_win function and self is: ', str(self))
```



Ex. 14.3 MainGUI Window

When the program runs and the “Print” and “Second Win” buttons are clicked, we see the same memory location for *self*. The output shows *self* in the main loop and it is passed to each of the functions.

```
In main and self is: <__main__.MainGUI object at 0x030B2330>
Self is now: <__main__.MainGUI object at 0x030B2330>
In second_win function and self is: <__main__.MainGUI object at 0x030B2330>
```

Ex. 14.3 Output

Next, a new window will be created from the call to *second\_win*, and *self* will not be implicitly passed. If the new window were created in a main function, *Toplevel* would be used to assign the parent window: – `sd_win = tk.Toplevel(self)`.

**Ex. 14.4** – Modifications to the MainGUI class to create a second window.

```
class SecondWin:
    def __init__(sd_win):

        print('In the second window and sd_win is: ', str(sd_win))

        sd_win = tk.Tk()
        sd_win.title('Second Window')
        sd_win.minsize(width=400,height=400)
        sd_win.configure(bg='white')

        sd_win.rowconfigure(0,minsize=100)
```

```
sd_win.header= tk.Label(sd_win, text='Second Window',\
                        font=('Helvetica',11))
sd_win.header.grid(row =0,column=1,columnspan=4)
```

The change to the *MainGUI* class is the addition of the *second\_win* function which creates the second window.

```
def second_win(self):
    print('In second_win function and self is: ', str(self))
    SecondWin = second_win.SecondWin()
```

When the program runs, and each of the buttons is clicked, the output shows the values of *self* and *sd\_win* and the window is created.

```
In main and self is: <__main__.MainGUI object at 0x03F92330>
Self is now: <__main__.MainGUI object at 0x03F92330>
In second_win function and self is: <__main__.MainGUI object at 0x03F92330>
In the second window and sd_win is: <second_win.SecondWin object at 0x04217F90>
|
```

Notice the memory location for *sd\_win* is different from the others. When an object is created, the name assigned is bound to the object. This means that if we were to add another line to *MainGUI* in the function *second\_win* (which creates the object), we should see the same memory as *sd\_win* when it is created.

```
def second_win(self):
    print('In second_win function and self is: ', str(self))
    SecondWin = second_win.SecondWin()
    print('SecondWin is: ', str(SecondWin))
```

```
In main and self is: <__main__.MainGUI object at 0x03F92330>
Self is now: <__main__.MainGUI object at 0x03F92330>
In second_win function and self is: <__main__.MainGUI object at 0x03F92330>
In the second window and sd_win is: <second_win.SecondWin object at 0x04416FB0>
SecondWin is: <second_win.SecondWin object at 0x04416FB0>
```

Ex. 14.4 Output



When the object is created, memory is allocated and the name *SecondWin* is bound to the object. Printing *SecondWin* displays the same location as *sd\_win*.

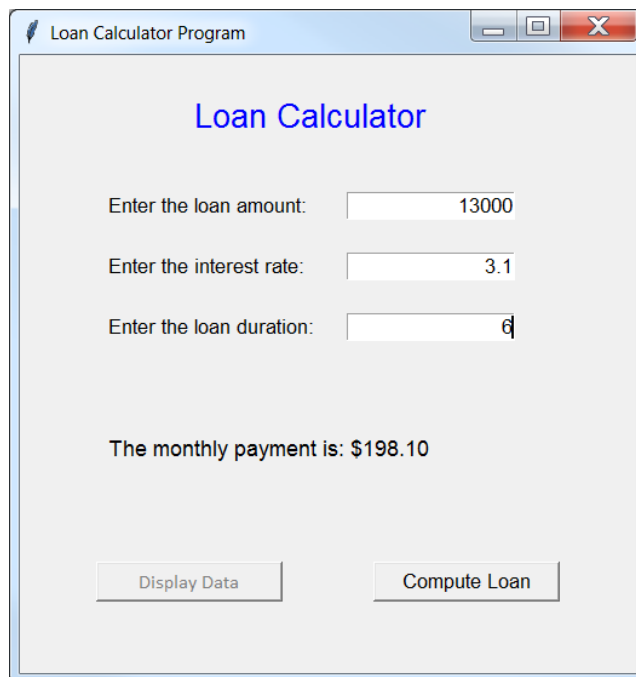
All of this helps to clear up the confusion associated with *self/root/master*. The second window could have just as easily used the name *self* as it did *sd\_win*. The idea is that a declaration like *sd\_win.header* is stating that the label control is to be associated with and located on the *sd\_win* window. The label belongs to *sd\_win*.

```
sd_win.header= tk.Label(sd_win, text='Second Window',\
                        font=('Helvetica',11))
```

The next step for data display requires that the main window send updates to the second window. Implementing this step will make things even clearer.

## Back to the Project

The project so far has a main GUI that responds to input and updates a *StringVar* on the main GUI with a computed value. At this point, the “Display Data” button is disabled. It will first need to create the second window and then be updated as the user enters additional input for computed values.



Loan Payment Example

The code below is for the Data Display window that will be created by the main window when the button is clicked. Recall that *self* will be passed implicitly to the function, but will not be used to create an object of the second window class.

#### Ex. 14.5 – Data Display Window

```
class DataDisplay:
    def __init__(sWin):

        sWin.dd_win = tk.Tk()
        sWin.dd_win.title('Loan Calculation History')
        sWin.dd_win.minsize(width=400,height=400)
        sWin.dd_win.configure(bg='white')

        sWin.dd_win.rowconfigure(0,minsize=50)

        # columns for Amount Interest Duration Payment
        for c in range(6):
            sWin.dd_win.columnconfigure(c, minsize=50)

        for r in range(1,10):
            sWin.dd_win.rowconfigure(r, minsize=15)

        # labels for Amount Interest Duration Payment
        sWin.dd_win.amount= tk.Label(sWin.dd_win, width=10, \
            text=' Amount', bg='white', font=('Helvetica',11))
        sWin.dd_win.header.grid(row=0,column=1)
```

The modifications to the main window code include: adding the command to the “Display Data” button, adding a flag as an indicator of whether or not the display window is open, modifying the *compute\_loan* function to update the display window, if it is open, and writing the function that will create the data display window and then disable the button. Ex. 14.6 shows these changes.

## Ex. 14.6 – Data Display Window Modifications

```

self.display_data_button = tk.Button(text='Display Data', width=16,\
                                     font=("Helvetica",10), command=self.display_data)
self.display_data_button.grid(row=8,column=1, columnspan=2)

self.display_win_open = False          # flag for display window

def compute_loan(self):                # compute_loan changes
    amt = self.amt_entry.get()
    -
    -
    -
    if self.display_win_open == True:  # flag for the display
        # Call display_data

def display_data(self):                # function to create the window
    self.DisplayDataWin = display_data_win.DataDisplay()
    self.display_data_button.config(state='disabled')
    self.display_win_open = True       # flip the flag

```

The creation of the window is the creation of an object of the `DataDisplay` class in a file called `display_data_win.py` and it is assigned to `self.DataDisplayWin`. When an object is created, it does not know about other objects unless it is told about other objects. In this case, the second window is created as an attribute of `self`, so that there is access to the window to perform updates to the display. The solution requires slight modifications to multiple parts of the program. Step-by-step design and walking through the process before adding the code helps to reduce omitted changes and saves time debugging the program.

The pseudo-code below walks through the steps for adding the display window. Notice the additional items included for Step 2. A flowchart could also surface steps that might otherwise be overlooked.

Display Data button click in pseudo-code:

- Step 1. The user clicks “Display Data”
- Step 2. The second window is created
  - The “Display Data” button is disabled
  - The `display_win_open` flag is set to True
- Step 3. The user clicks the “Compute” button
- Step 4. The `display_data` function is called

There are a few different design solutions that could be used to display the data in the second window. The solution chosen here creates a new label for each data item and adds it to the second window, incrementing the column and row. Since there is a “handle” to the window through *self*, a label can be created and a grid location can be assigned from the `display_data` function.

Ex. 14.7 – Updating the Data Display Window

```

self.row_count = 2                # added to main to be used for grid

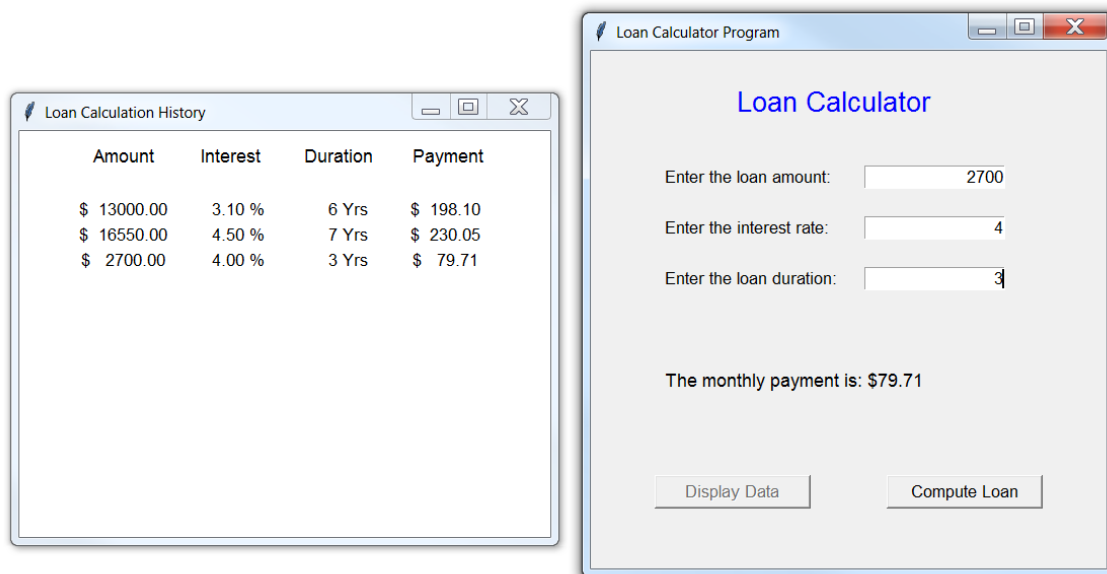
def display_data(self):           # modifications to display_data
    if self.display_win_open == False:
        self.DisplayWin = display_data_win.DataDisplay()
        self.display_win_open = True
        self.display_data_button.config(state='disabled')
    else:
        fltA = float(self.amt)     # convert the values to float
        fltI = float(self.intst)
        intD = int(self.dur)       # convert to integer
        fltMP = float(self.mp)
        self.DisplayWin.dd_win.amt_lbl = tk.Label(self.DisplayWin.dd_win, \
            font=('Helvetica',10), bg='white', \
            text='$' + str(format(fltA, '10.2f')))
        self.DisplayWin.dd_win.amt_lbl.grid(row=self.row_count,column=1)

```

A new variable is added to the class called *row\_count* to be used for grid placement of the labels in the second window. It is incremented each time the function is called. The *display\_data* function is modified to include an else clause to do the updating. The values are converted in order to format them properly, and then they are converted back to strings for display as shown here.

```
fltA = float(self.amt)           # convert to float
text= '$' + str(format(fltA, '10.2f')) # format and convert to string
```

Each time the “Compute Loan” button is clicked, the display updates with the new values by creating a label with the text and placing it on the grid in a new row.



Ex. 14.7 Display Output

## File Data Display

Reading data from a file and displaying it could be handled in a similar way. For example, assume that there is a data file containing a data set that would be displayed in five columns. A loop will read from the file and display the data by creating the labels as the values are read. They could also be read into a list or tuple and again a loop would be used to create the labels. Since the data in this case is not used in a computation, a single loop creation can be used and a row/column algorithm can be used for the grid placement.

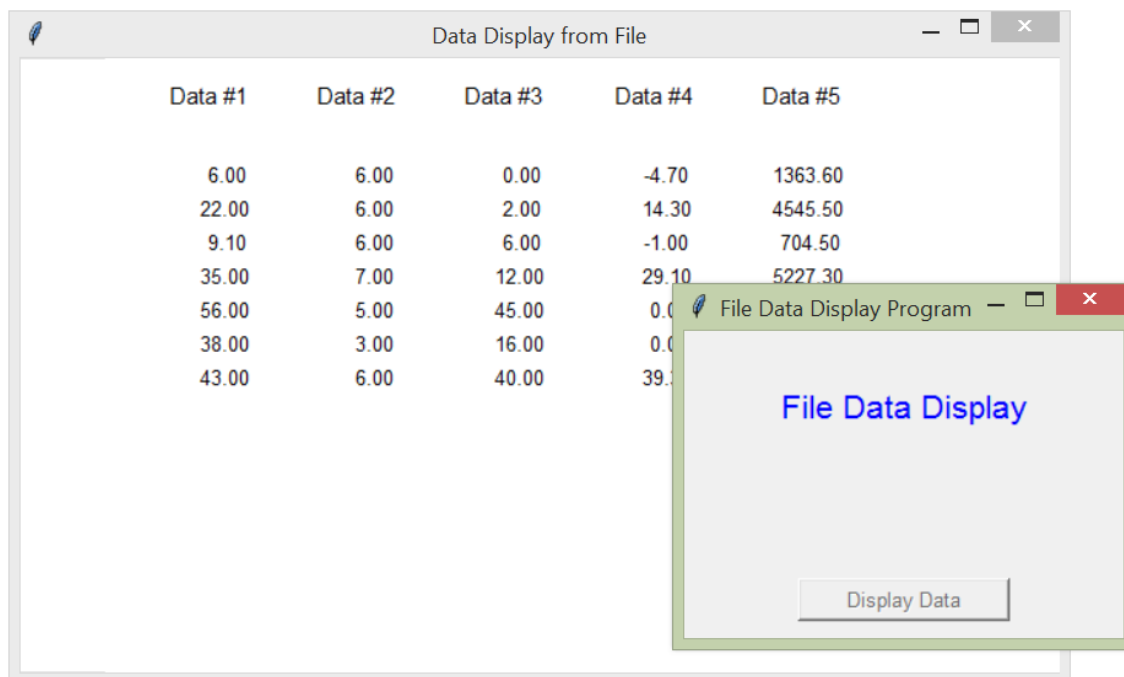
Ex. 14.8 – Updating the Data Display Window from a file.

```

for value in inFile:
    fltVal = float(value)
    self.DisplayWin.dd_win.data_lbl = tk.Label(self.DisplayWin.dd_win,\
        font=('Helvetica',10), bg='white', text= str(format(fltVal, '10.2f')))\
        .grid(row=self.r_count,column=self.c_count)

self.col_count = self.c_count + 1
if self.col_count == 6:
    self.col_count = 1
    self.row_count = self.r_count + 1

```

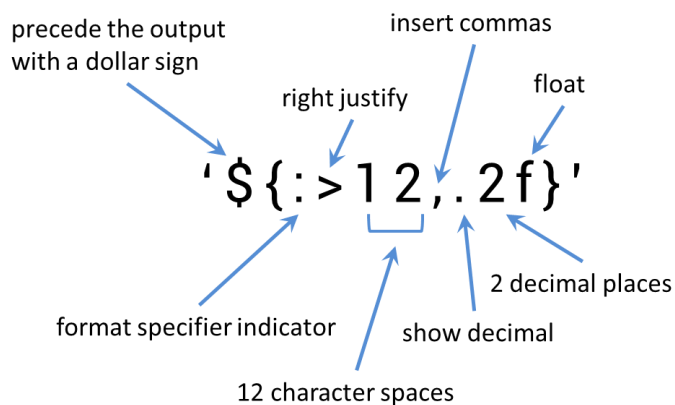


Ex. 14.8 Display Output

## Formatting Data

In addition to the format specifiers shown previously, Python has introduced a new version that is often simpler to use. The formatting is surrounded by braces.

```
fltA = float(amt)          # convert the value to float
fltA_string = '${:>12,.2f}'.format(fltA)
```



## Plotting Data

Data can also be plotted to a display window by drawing on a canvas control. The following example computes a fahrenheit temperature from a Celsius input and plots both values in a separate display. The GUI code for the window and controls is omitted. The function *convert* plots the values to a canvas using *x,y* coordinates as the first arguments to *create\_text* and *create\_oval*, and changes the color for each to blue for the negative values.

Ex. 14.9 – Plotting using a Canvas.

```
def convert(self):
    self.celsius = float(self.celsius_entry.get())
                                # Calculate Fahrenheit
    self.fahrenheit = round((9.0 / 5.0 * float(self.celsius)) + 32, 2)
    self.fahr.set(self.fahrenheit)    # Update the fahrenheit_label

    color = 'black'                # determine the text color
    if self.fahrenheit < 0:
        color = 'blue'

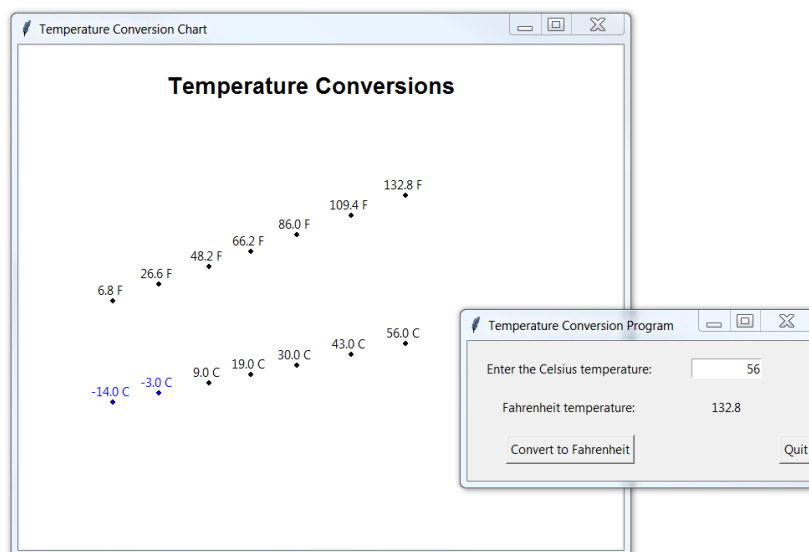
                                # populate the canvas with the value
    self.canvas.create_text(180 + self.celsius*5, 300-self.fahrenheit, \
                            fill=color, text= str(self.fahrenheit)+ ' F')
                                # draw the oval
    self.canvas.create_oval(180 + self.celsius*5, 310-self.fahrenheit, \
                            185 + self.celsius*5, 315-self.fahrenheit, fill = color)
```

```

color = 'black'
if self.celsius < 0:
    color = 'blue'
                                # populate the canvas with the value
self.canvas.create_text(180 + self.celsius*5, 400-self.celsius, \
                        fill = color, text=str(self.celsius)+ ' C')
self.canvas.create_oval(180 + self.celsius*5, 410-self.celsius, \
                        185 + self.celsius*5, 415-self.celsius, fill = color)

```

The program displays the text and marker as the button is clicked.



Ex. 14.9 Plotting Output

Another Python module for plotting data is matplotlib. The matplotlib library is extensive and can be used to generate plots, histograms, bar charts, scatterplots, and more. Chapter 15 includes an example using matplotlib.



# Chapter 15

## Python Modules

One of the benefits to programming in Python is the extensive list of modules that have been developed over the years. A Python library is a collection of functions and methods that can be imported and used without development. As an example, the Python imaging library (PIL), can be used for manipulating images. For real-time vision and image processing, the library Open-CV, which also binds to C++, C#, and others, can be imported.

The number of libraries is too extensive to list, but Python developers seem to agree on a few that require mentioning. In alphabetical order:

Beautiful Soup - XML and HTML parsing library

Bokeh - support large-scale interactivity and visualizations of real-time data sets

Karas - neural networks API, supports deep learning

NLTK - language processing, string manipulation

Nose - testing framework for Python

Numpy - math functions

matplotlib - data visualization, a numerical plotting library (example below)

Pandas - data manipulation and analysis

Pillow - imaging library

Plotly - publication-quality plots and graphs, finance and geospatial industries

Pygame – 2D game development  
Pyglet – 3D animation and game creation engine  
pyGtk - Python GUI library  
pyQT - GUI development  
PyTorch - neural network modeling library with GUI  
pywin32 - interaction with Windows  
IPython - shell for Python  
Requests - HTTP library  
SymPy - algebraic evaluation, complex numbers, differentiation  
Scrapy - extract data and a web crawler  
SciPy - algorithm and mathematical tools, signal processing, optimization and statistics  
Scikit-learn - machine learning and data mining  
SpaCy - large scale extracting/analysis of textual information; Supports deep learning  
SQLAlchemy - database library  
Tensorflow - machine learning and deep learning  
Twisted - networking applications development  
wxPython - GUI toolkit

## Plotting with Matplotlib

The matplotlib library module pyplot provides functionality to generate line, bar, and pie charts in an auto-scaling resizable window. The next example uses matplotlib to generate a line graph of sales data that has been read in from a file. The data is stored in a list and is passed to a function called *plot\_mylist* which sets up the window, the chart, and it also plots the data. Note the import statement for pyplot requires the library and dot.

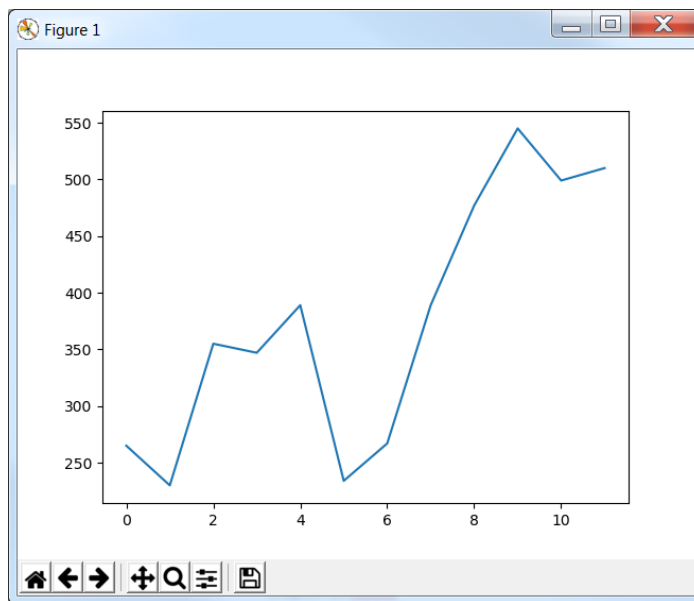
```
import matplotlib.pyplot as plt
```

There are many options available for customizing the charts including: axis labels, tick marks, data markers, the width of bars, and slice labels for pie charts. The next example uses a few of these.

The function `plot_mylist` shown below first establishes the number of data points for each axis using `x_coords` and `y_coords`. The number of x-axis tick marks is established from the number of coordinates in the list. The number of y-axis tick marks and their values are established by the values in the `sales_list` data. The call to `plt.plot` actually builds the graph in memory, and it is then displayed when `plt.show()` is called.

#### Ex. 15.1 – Initial Line Graph of Sales Data using pyplot

```
def plot_mylist(sales_list):
    x_coords = [0,1,2,3,4,5,6,7,8,9,10,11]
    y_coords = sales_list
    plt.plot(x_coords, y_coords)
    plt.show()
```



Ex. 15.1 Initial Pyplot Output

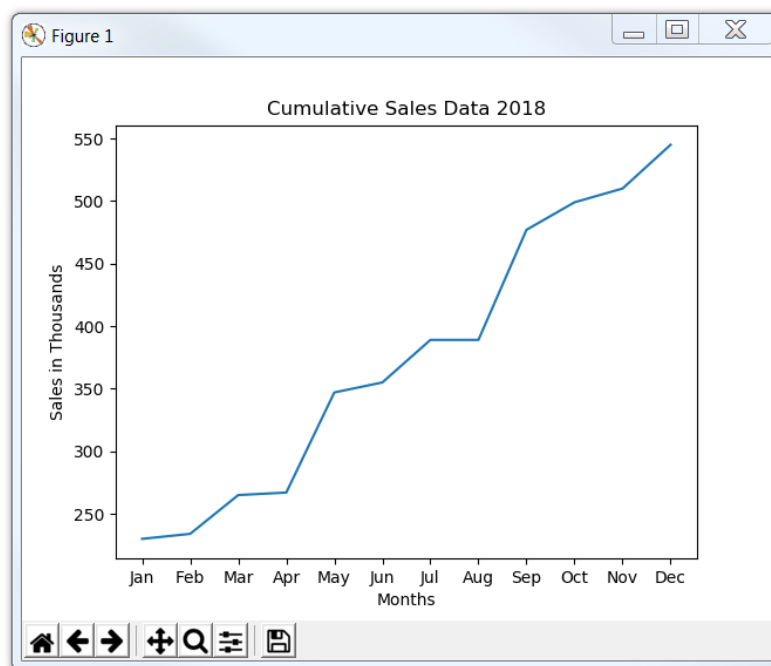
The data is plotted and the window has many features included in the lower left-hand corner for zooming in a rectangular shape, saving the image, and others.



The graph is complete, but it is missing information that would clarify what is being displayed. Some of the additional features of pyplot include axis labels, tick mark labels, and a title for the chart.

#### Ex. 15.2 – Line Graph of Sales Data Enhanced

```
def plot_mylist(sales_list):  
    x_coors = [0,1,2,3,4,5,6,7,8,9,10,11]  
    y_coors = sales_list  
    plt.xticks([0,1,2,3,4,5,6,7,8,9,10,11], ['Jan', 'Feb', 'Mar', 'Apr', 'May',  
                                                'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'])  
    plt.title('Cumulative Sales Data 2018')  
    plt.ylabel('Sales in Thousands')  
    plt.xlabel('Months')  
    plt.plot(x_coors, y_coors)  
  
plt.show()
```

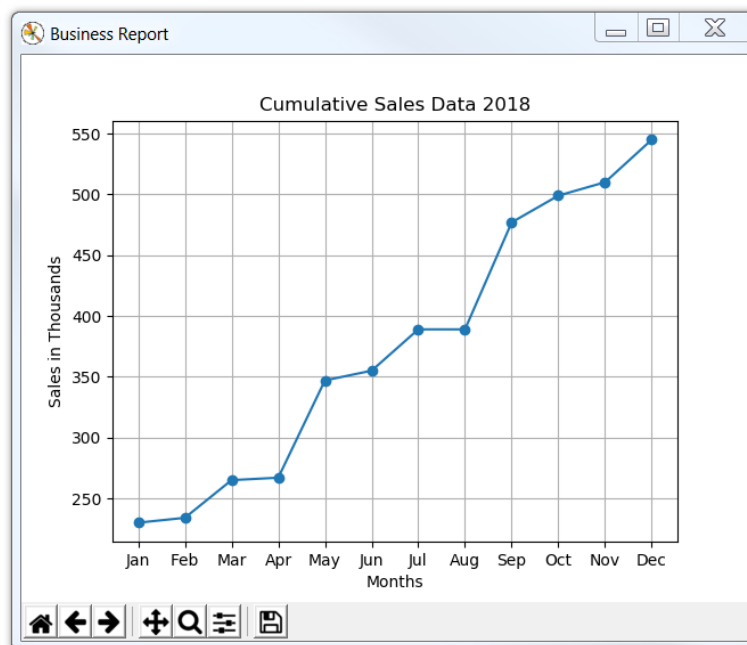


Ex. 15.2 Pyplot Output

Other enhancements might include changing the window title on the title bar, adding shapes to the data points, and adding a background grid to the chart.

### Ex. 15.3 – Line Graph of Sales Data Complete

```
def plot_mylist(sales_list):
    x_coords = [0,1,2,3,4,5,6,7,8,9,10,11]
    y_coords = sales_list
    plt.xticks([0,1,2,3,4,5,6,7,8,9,10,11], ['Jan', 'Feb', 'Mar', 'Apr', 'May',
                                                'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'])
    fig = plt.gcf() # get current figure
    fig.canvas.set_window_title('Business Report')
    plt.title('Cumulative Sales Data 2018')
    plt.ylabel('Sales in Thousands')
    plt.xlabel('Months')
    plt.grid(True) # grid background
    plt.plot(x_coords, y_coords, marker='o') # data point markers
    plt.show()
```



Ex. 15.2 Pyplot Output

Modules, like `pyplot`, are easy to use and provide extensive functionality. To generate the sales data chart and resizable chart window only required eleven lines of code.

# Chapter 16

## File Dialogs, HTML, and Animation

This chapter contains additional features and functionality in Python including; producing sound, animation, opening a browser, and creating a file open dialog. In many cases, the key is determining the module to use and the options available. There is also the confusion created by changes in the language such as the fact that Python 3.0 is not backward compatible, and there is a great deal of information available that does not specify the version of Python being used.

### File Dialogs

As an example, an Open File dialog is available in Python 2 (Ex. 16.1A), but does not work correctly in Python 3.0. The example in Ex. 16.1A does.

**Ex. 16.1** – Open File Dialog Example – Python 2

```
import Tkinter, tkFileDialog

self = Tkinter.Tk()
self.withdraw()                # hide the root window

file_path = tkFileDialog.askopenfilename()
```

Compare the module imported and the code in the lines above to those below.

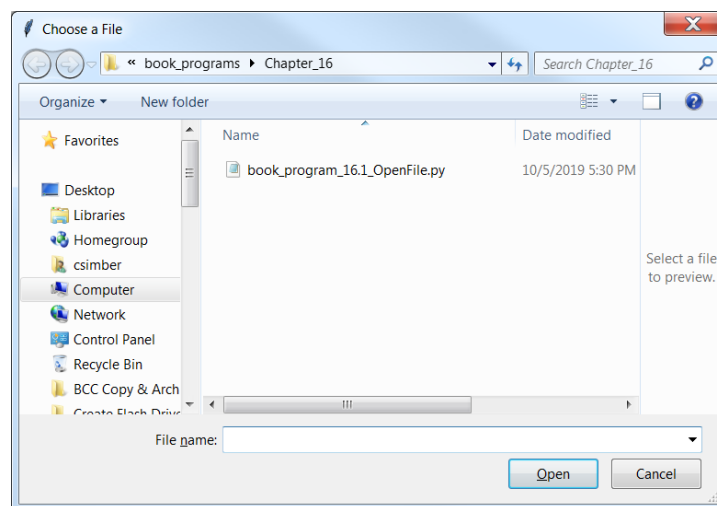
Ex. 16.1A – Open File Dialog Example – Python 3.9.0.

```
import tkinter as tk
from tkinter import filedialog

self = tk.Tk()
self.withdraw() # hide the root window

file_path = filedialog.askopenfilename(title='Choose a File')
```

The *self* or root window is withdrawn in this program so that only the dialog appears and the default directory is the directory where the program is running.



Ex. 16.1A Open File Dialog

To create a Save As dialog, a change to one line of code is necessary.

```
file_save = filedialog.asksaveasfilename()
```

To predetermine file types, a list of types with descriptions is created and is passed to the dialog which is used in the drop-down list as shown below.

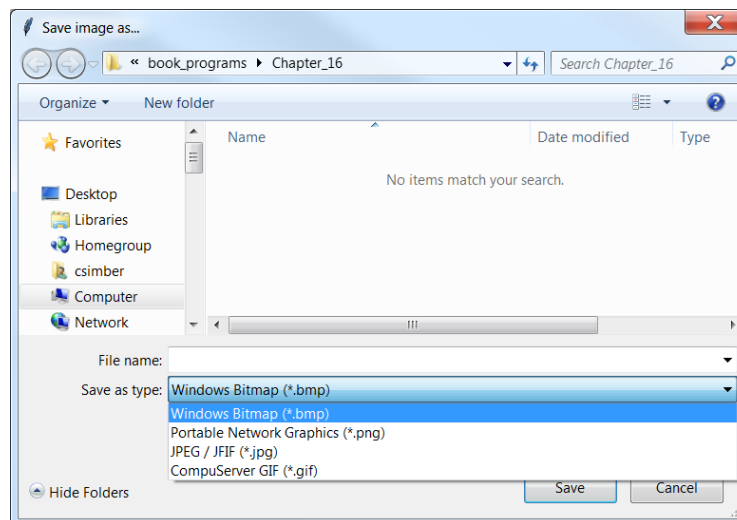


The example below creates a list of image file types with descriptions for the dialog.

#### Ex. 16.2 – Save As Dialog with File Types

```
myFormats = [ ('Windows Bitmap', '*.bmp'),
              ('Portable Network Graphics', '*.png'),
              ('JPEG / JFIF', '*.jpg'), ('CompuServer GIF', '*.gif'), ]
```

```
file_types= filedialog.asksaveasfilename(filetypes= myFormats,
                                         title = 'Save image as...')
```



Ex. 16.2 Save As Dialog with File Types

## HTML and Browsers

Browsers read \*.html files and Python can be used to create the html file and then launch the browser with the file. In the following example, the Python program creates a file. The program then writes some standard HTML to the file, closes it, and then launches a browser with the file.

Note the import statement for *webbrowser* and that the HTML to be written to the file is surrounded by triple quotes and assigned to *message*.

### Ex. 16.3 – Create an HTML File and Launch a Browser

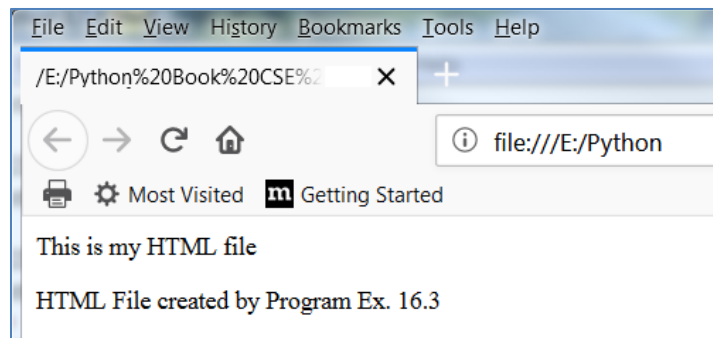
```
import webbrowser

f = open('my_html_file.html','w')

message = """<html>
<head>This is my HTML file</head>
<body><p>HTML File created by Program Ex. 16.3</p></body>
</html>"""

f.write(message)
f.close()

webbrowser.open_new_tab('my_html_file.html')
```



Ex. 16.3 Web Page created by Python program

This simple example of creating HTML and launching a browser can be augmented extensively to create complex web pages including updates and the use of variables.

## Animation

Python can be used to generate animation by creating a series of frames and assembling them with *makeMovieFromInitialFile* similar to the way an MPEG file is merged into a sequence of images. Matplotlib's animation class also provides this capability with *FuncAnimation*, which repeatedly calls a function. Using it requires importing the numpy module and matplotlib's animation module.

The FuncAnimation interface accepts arguments for:

- fig** - the Figure Object used
- func** - the callable function at each frame
- frames** – source of data to pass to func for each call (number of frames)
- init\_func** – used to draw a clear frame (optional)
- fargs** - tuple or none (additional callable function arguments) (optional)
- save\_count** - number of frames to cache (optional)
- interval** - number (optional), the delay between frames in milliseconds
- repeat\_delay** - number in milliseconds (optional) between repeats
- repeat** - boolean (optional), should the animation repeat
- blit** – boolean for blitting to optimize drawing to reduce time
- cache\_time\_data** – boolean for whether frame data is cached

**Ex. 16.4** – Carrier Wave Animation. 94.4 MHz

1. `import numpy as np`
2. `from matplotlib import pyplot as plt`
3. `from matplotlib import animation`
- 4.
5. `fig = plt.figure()`
6. `fig.canvas.set_window_title('Carrier Wave Analysis')`

Lines 1 thru 3 import numpy, pyplot, and animation

Line 5 creates the figure window for display and line 6 sets the window title

```

7. thismanager= plt.get_current_fig_manager()
8. img = PhotoImage(file='Radio.gif')
9. thismanager.window.tk.call('wm', 'iconphoto', \
    thismanager.window._w, img)

```

Line 7 gets a window manager (handle) for the icon assigned to the image on line 8, and then line 9 adds the icon to the window.

```

10. ax = pltaxes(xlim=(0,5), ylim=(0,1))
11. line, = ax.plot([],[]), lw=4)
12. def init():
13.     line.set_data([],[])
14.     return line

```

Line 10 sets up the x and y axis and sets their limits

Line 11 creates an empty line object (filled later), and sets the line width

Line 12 defines the init function that will be passed to FuncAnimation

Line 13 sets the data for the line (lists of x and y values)

Line 14 returns the line

```

15. def animate(i):
16.     x = np.linspace(0, 5, 1000)
17.     plt.xlabel('Milliseconds')
18.     plt.ylabel('MHz')
19.     plt.title('Carrier Wave Over Time')
20.     plt.yticks([0,1,2,34], ['94.0', '94.2', '94.4', '94.5', '94.6'])
21.     y = ((np.sin(2*np.pi * (x-0.01*i)) * 0.1) +2)
22.     line.set_data(x,y)
23.     return line

```

Line 15 defines the animate function which is passed to FuncAnimation

Line 16 sets up the numpy line space start point, end point, and number of samples to generate

Line 17 and 18 assign the x and y axis labels

Line 19 sets the title for the chart

Line 20 establishes the number of y tick marks and their labels

Line 21 is the equation for generating the wave

Line 22 assigns the x and y values to the line lists

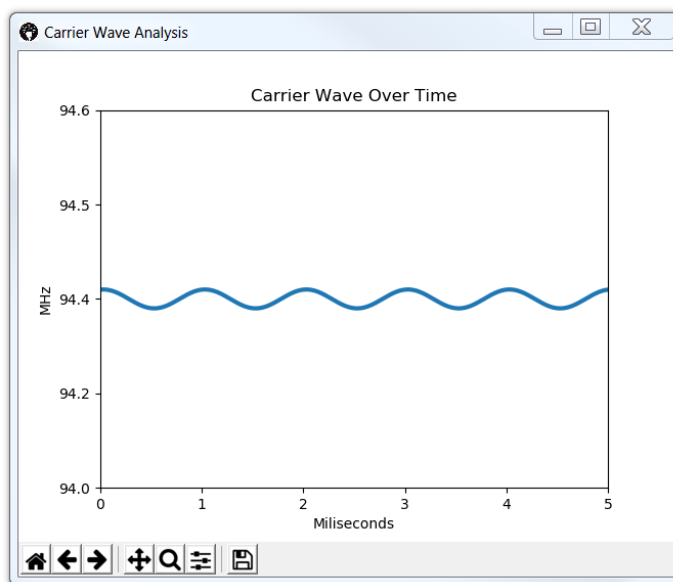
Line 23 returns the updated line

```

24. anim = animation.FuncAnimation(fig, animate, init_func=init,\
25.     frames = 200, interval=20, blit=False)
26.
27. plt.show()

```

Lines 24 and 25 are the call to animate passing the figure object, the animate function (callable at each frame), the init function (draws a clear frame), the number of frames, the interval or delay between frames in milliseconds, and no blitting.



Ex. 16.4 Animated Carrier Wave

## Complete code for Ex. 16.4

```

import numpy as np
from matplotlib import pyplot as plt
from matplotlib import animation
from tkinter import PhotoImage

fig = plt.figure()
fig.canvas.set_window_title('Carrier Wave Analysis')
thismanager = plt.get_current_fig_manager()
img = PhotoImage(file='Radio.gif')
thismanager.window.tk.call('wm', 'iconphoto', thismanager.window._w, img)

ax = plt.axes(xlim=(0,5), ylim=(0,1))
line, = ax.plot([],[],lw=4)

def init():
    line.set_data([],[])
    return line

def animate(i):
    x = np.linspace(0,5,1000)
    plt.xlabel('Milliseconds')
    plt.ylabel('MHz')
    plt.title('Carrier Wave Over Time')
    plt.yticks([0,1,2,3,4], ['94.0', '94.2', '94.4', '94.5', '94.6'])
    y = ((np.sin(2* np.pi * (x-0.01*i)) * 0.1) + 2)
    line.set_data(x,y)
    return line

anim = animation.FuncAnimation(fig, animate, init_func=init,
                               frames=200, interval=20, blit=False)

plt.show()

```

'Blitting' (Bit Block Transfer) is an old technique in computer graphics. The idea is to take an existing image and then 'blit' another on top of it instead of erasing each pixel and redrawing them. By managing a saved 'clean' bitmap, the program only re-draws the few areas that are changing at each frame. This often saves significant amounts of time redrawing. To use blitting, set `blit=True` in the arguments to `FuncAnimation`.

## Sound

On Windows®, `winsound` provides standard sound playing capability and the Windows sound files. For other operating systems, there are a variety of modules that can be installed with the PIP installer.

### Ex. 16.5 – Playing Audio

```
import winsound

def main():

    winsound.PlaySound("SystemExclamation",winsound.SND_FILENAME)
    winsound.PlaySound("SystemHand", winsound.MB_OK)
    winsound.PlaySound('my_own.wav', winsound.SND_FILENAME)

main()
```

Other sound modules available for Python include `playsound`, `pygame`, `pyMedia`, `pyglet`, and more. Any of these can be installed with PIP to import and use for playing MP3 files and most audio formats.



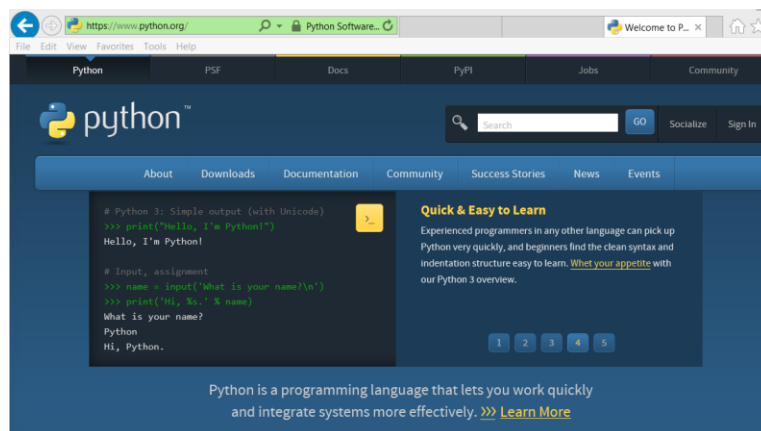


## Appendix A

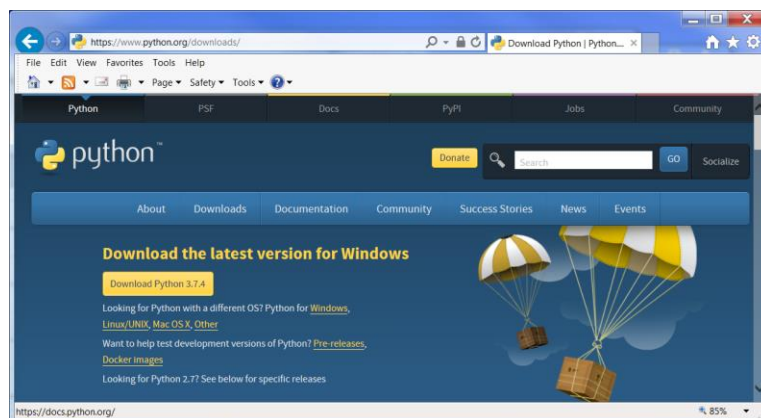
### Obtaining Python with IDLE

- Python and IDLE can run on any machine, and can be installed and runs fine on a flash drive
- The IDLE IDE is installed with Python 3.7.1 and above
- The tkinter module is installed with Python
- Python is available from Python.org <https://www.python.org>

Browse to the Python web site shown here and select “Downloads”.



In the Downloads window shown below, select the “Download Python 3.7.4” button or select as appropriate for your computer. A later version may be available.



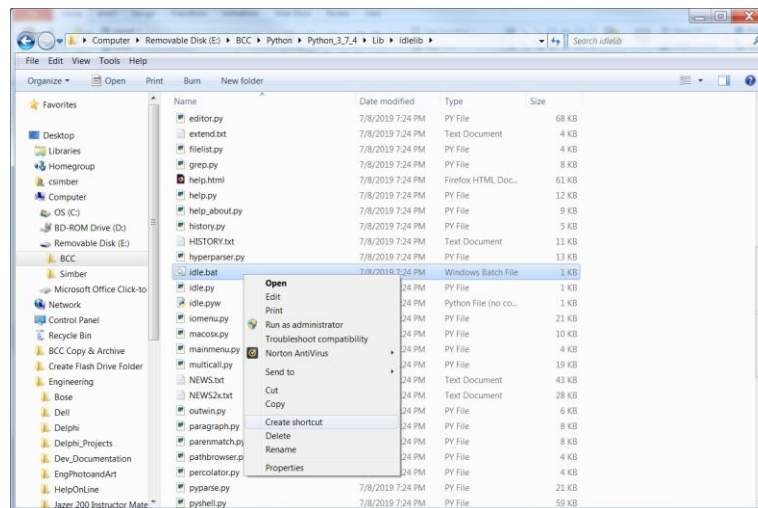
Select the folder where to install the program and download or save it to a folder.

## Appendix A

Python and IDLE run fine on a flash drive and can be installed there if you prefer. The folders and files shown below are installed with Python.

Name	Date modified	Type	Size
DLLs	7/20/2019 3:55 PM	File folder	
Doc	7/20/2019 3:58 PM	File folder	
include	7/20/2019 3:55 PM	File folder	
Lib	7/20/2019 3:55 PM	File folder	
libs	7/20/2019 3:55 PM	File folder	
Scripts	7/20/2019 4:00 PM	File folder	
tcl	7/20/2019 3:58 PM	File folder	
Tools	7/20/2019 3:58 PM	File folder	
LICENSE.txt	7/8/2019 7:33 PM	Text Document	30 KB
NEWS.txt	7/8/2019 7:33 PM	Text Document	676 KB
python.exe	7/8/2019 7:31 PM	Application	96 KB
python-3.7.4.exe	7/20/2019 3:45 PM	Application	25,063 KB
python3.dll	7/8/2019 7:30 PM	Application extens...	58 KB
python37.dll	7/8/2019 7:29 PM	Application extens...	3,522 KB
pythonw.exe	7/8/2019 7:31 PM	Application	94 KB
vcruntime140.dll	7/8/2019 7:24 PM	Application extens...	85 KB

**Note:** The IDLE executable is not at this level. It is in Lib/idlelib and is called idle.bat. Double clicking idle.bat will launch the IDE. To simplify launching IDLE each time, creating a shortcut is recommended. In some cases a desktop shortcut may have been installed when the program was installed.



IDLE is launched by double-clicking: Lib\idlelib\idle.bat

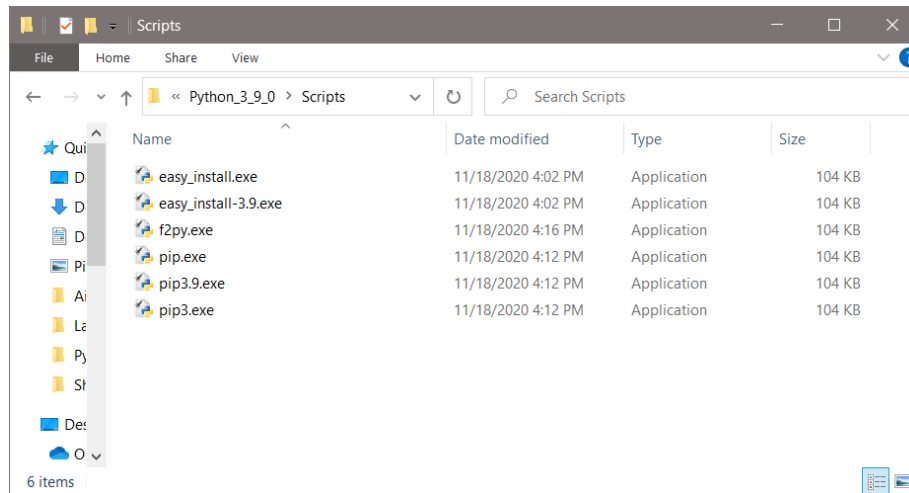
Documentation can be found at:

<https://docs.python.org/2/library/idle.html>

## Appendix B

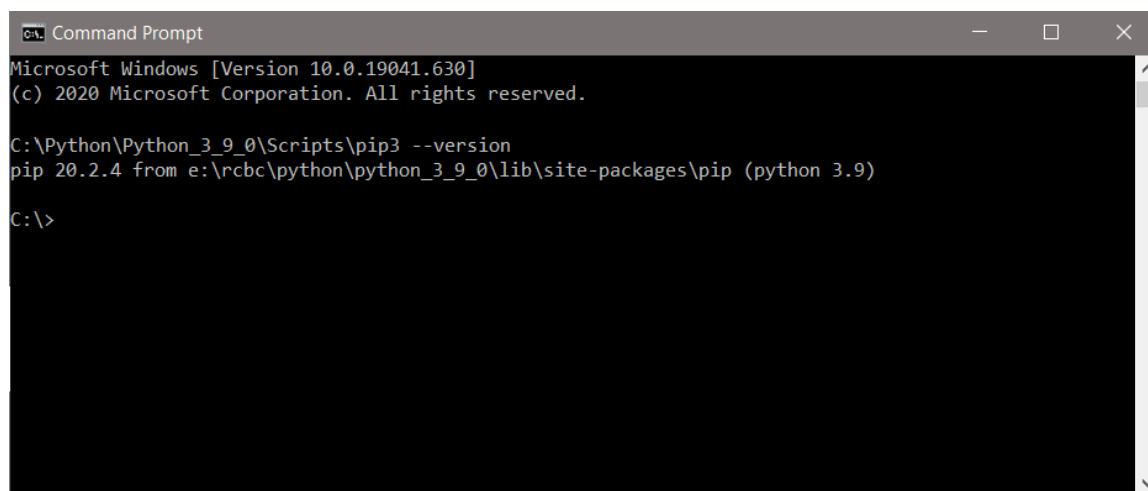
### The PIP Module Installer

PIP is already installed if you are using Python 3.4 or above. PIP is a command line program (no GUI), and is run typically from a command prompt. To confirm that PIP is installed, open the Python folder and then Scripts folder which will include PIP files.



PIP can also be verified by opening the Python folder and then the Lib folder, and then the site-packages folder. The PIP installer is run from the Scripts directory. To test for PIP and the proper directory, open a command prompt and type the path to python\Scripts and then pip3 then minus minus version. In this example, Python 3.9.0 is installed (which comes with pip3). The directory is Python\Python\_3\_9\_0.

```
C:\Python\Python_3_9_\Scripts\pip3 --version
```



```
Microsoft Windows [Version 10.0.19041.630]
(c) 2020 Microsoft Corporation. All rights reserved.

C:\Python\Python_3_9_0\Scripts\pip3 --version
pip 20.2.4 from e:\rcbc\python\python_3_9_0\lib\site-packages\pip (python 3.9)

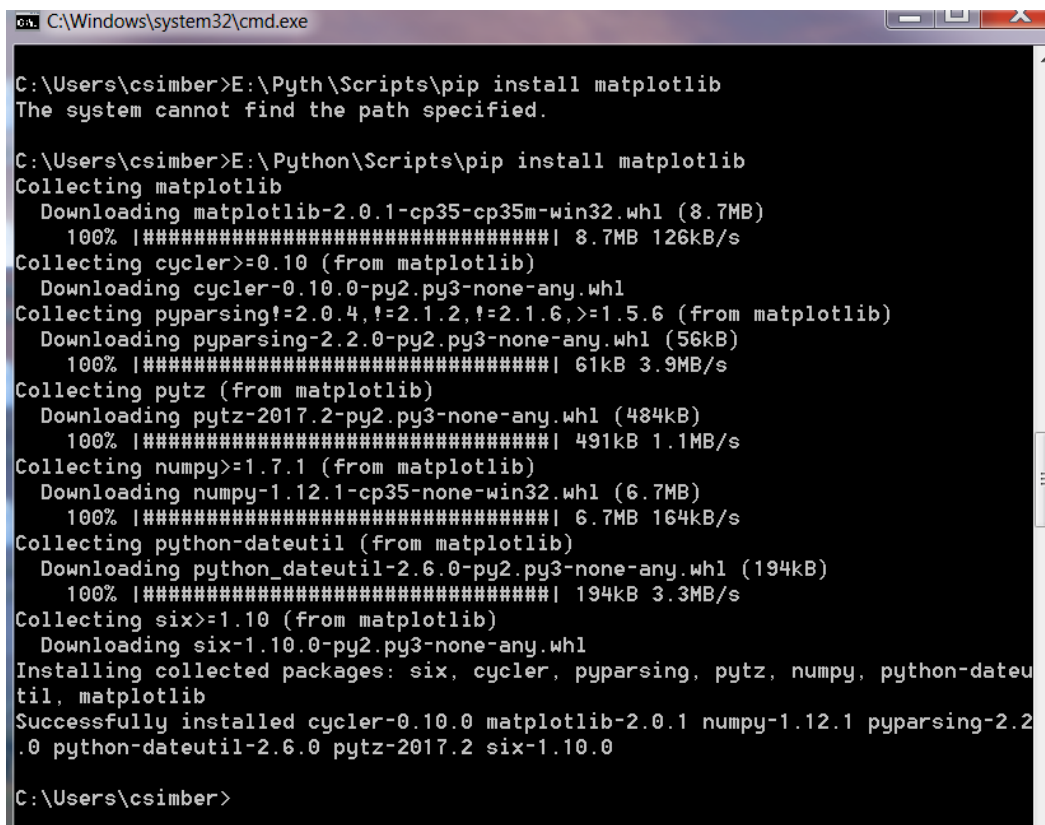
C:\>
```

## Appendix B

It is important to use complete paths when using PIP, and to be careful of typographical errors. The example below is installing the matplotlib module which is a Python plotting tool. It installs the module using PIP which is being run from the following subdirectory:

E:\Python\Scripts

The first command shown below omitted the letters “on” from “Python” in the command. The second successfully used the installer.



```

C:\Windows\system32\cmd.exe

C:\Users\csimber>E:\Pyth\Scripts\pip install matplotlib
The system cannot find the path specified.

C:\Users\csimber>E:\Python\Scripts\pip install matplotlib
Collecting matplotlib
  Downloading matplotlib-2.0.1-cp35-cp35m-win32.whl (8.7MB)
    100% |#####| 8.7MB 126kB/s
Collecting cyclер>=0.10 (from matplotlib)
  Downloading cyclер-0.10.0-py2.py3-none-any.whl
Collecting pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=1.5.6 (from matplotlib)
  Downloading pyparsing-2.2.0-py2.py3-none-any.whl (56kB)
    100% |#####| 61kB 3.9MB/s
Collecting pytz (from matplotlib)
  Downloading pytz-2017.2-py2.py3-none-any.whl (484kB)
    100% |#####| 491kB 1.1MB/s
Collecting numpy>=1.7.1 (from matplotlib)
  Downloading numpy-1.12.1-cp35-none-win32.whl (6.7MB)
    100% |#####| 6.7MB 164kB/s
Collecting python-dateutil (from matplotlib)
  Downloading python_dateutil-2.6.0-py2.py3-none-any.whl (194kB)
    100% |#####| 194kB 3.3MB/s
Collecting six>=1.10 (from matplotlib)
  Downloading six-1.10.0-py2.py3-none-any.whl
Installing collected packages: six, cyclер, pyparsing, pytz, numpy, python-dateutil, matplotlib
Successfully installed cyclер-0.10.0 matplotlib-2.0.1 numpy-1.12.1 pyparsing-2.2.0 python-dateutil-2.6.0 pytz-2017.2 six-1.10.0

C:\Users\csimber>

```

If running from a command console is an issue, PIP can be run through the Python interpreter. The complete User Guide for PIP is available at:

[https://pip.pypa.io/en/stable/user\\_guide/](https://pip.pypa.io/en/stable/user_guide/)

## Appendix C

### The Data Analysis Project

This project develops an interactive Python GUI program that extracts various weather data elements from a file for correlation, display, and plotting.

When the program launches, a login/create account window will open first. When the create account button is clicked, a separate window will be created from a separate GUI class in a separate module. The user must create a unique username and a password that is nine (9) characters or more, with at least one digit, uppercase and lowercase letter. The program will continue to show error messages and prompt for a password until a valid password is created. The valid account information will be stored for retrieval for future login. When an account has been created and the login button is clicked, the main window can change to accept the user name and password or another window can be used. The user name and password will be checked against existing accounts to ensure that the account exists.

When a valid login has occurred, the main window will change to allow selection of the data to display. The user will be able to select one of four (4) different data sets from a National Weather Service file (provided), and select a time interval for data extraction. The data sets to choose from are:

- Barometric pressure and temperature
- Barometric pressure and wind speed
- Barometric pressure and sky cover
- Temperature and dew point

The time interval for data extraction can be any whole number of years' worth of data from 2010 through 2018. Once a data set and time window have been selected, a display window will open with column headers and the data vertically in columns with the capability for the user to plot the data. Plotting functionality should be disabled until the user has selected the data to be plotted.

The data in the file is columnar with no delimiter and is accompanied by data dictionary. A sample of the data format is shown here.

```
File Edit Format View Help
YR--MODAHRMN DIR SPD GUS CLG SKC L M H VSB MW MW MW AW AW AW AW W TEMP DEWP SLP ALT STP MAX MIN PCP01
201001010011 *** 0 *** 5 OVC *** 3.0 ***** 10 ***** 34 30 ***** 30.07 1016.4 ***** ***** *****
201001010041 *** 0 *** 5 OVC *** 2.5 ***** 10 ***** 34 30 ***** 30.07 1016.4 ***** ***** *****
201001010054 *** 0 *** 5 OVC *** 3.0 ***** 41 40 10 *** 32 30 1018.2 30.07 1016.4 *** ** 0.00T***** *****
201001010117 *** 0 *** 5 OVC *** 2.5 ***** 41 40 10 *** 32 30 ***** 30.06 1016.0 *** ** 0.00T***** *****
201001010133 340 3 *** 43 OVC *** 2.0 ***** 41 40 10 ** 32 30 ***** 30.06 1016.0 *** ** 0.00T***** *****
```

A **Design Document** is required and will be submitted for odd numbered Milestones, presented as a part of project demonstrations for even Milestones, and submitted as a final submission. The Design Document will include step-by-step implementation screen captures, descriptions, and explanations of functionality in the program (see the sample file), followed by the program code.

**Presentations** are required during the semester and a final **Presentation/demonstration**. The Design document will be presented first during each presentation including the program code, and then the program operation will be demonstrated.

## Appendix C

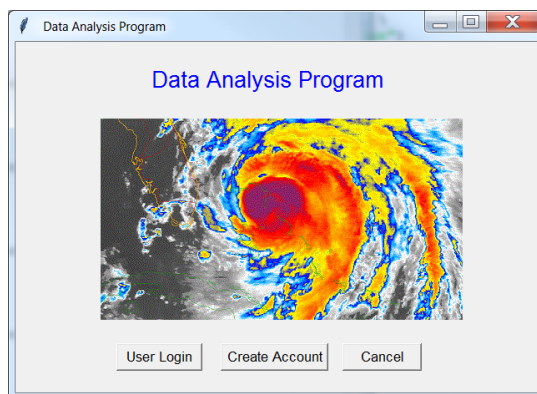
*The screen captures in this document are for reference only. The design and interface of your program do not have to mirror the examples in this document in terms of appearance and controls used, but must handle the operations.*

### Assignment

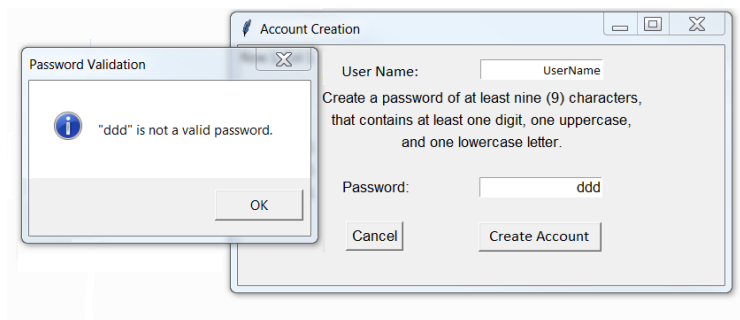
Get Python with IDLE and create a working program using the “Getting Started in Python” exercise. The latest IDLE comes with Python 3.9.0 from python.org. Determine the installation location. IDLE will run fine on a flash drive. Create and run a “Hello World”-like program. Create a design/development document that you will update when you work the project. This will be reviewed at the Milestones as will the code. See the example design documentation.

### Project Milestone #1

Begin the **Design/development** documentation with a general description of the project and include screen captures and explanations of operation for the milestone. Create the main GUI loop for the program and the create account interface using buttons, and entry controls.



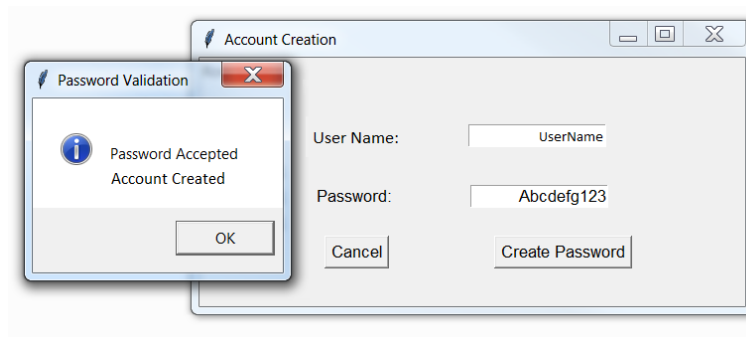
Design and implement the Account Creation GUI using a class in a separate file that is launched when the “Create Account” button is clicked. Design and develop the functionality for retrieving the username and validating the password. Right-align the input text and create an error dialog that echoes the password. Validate the password for length (9 characters or more), an uppercase and lowercase letter, and a digit. The window must contain instructions and operating “Cancel” and “Create Account” buttons.



## Appendix C

When the “OK” button is clicked, the previous password should be cleared.

```
password_entry.delete(0,END) # clear the text
```



Note: The “Create Account” button should create an instance of the Account Creation class.

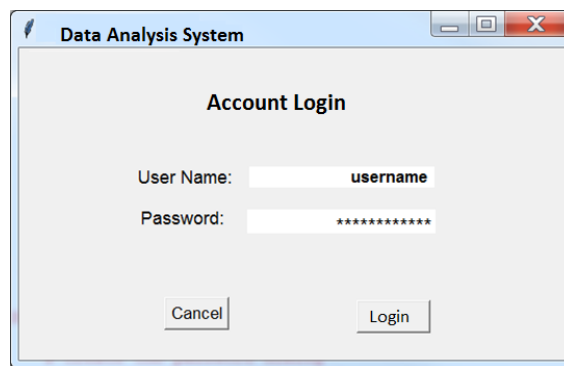
Continue the username and password algorithm by saving the data and storing it in a file for retrieval by the login function. The password stored should be encrypted. This can be done later in the project. When account creation is successful, the “Create Account” button on the main GUI should be disabled since an account was just created.

**STATUS:** At this point, there should be a main loop for the program with the main interface. There should be a separate file with a class that handles the creation of a user account that contains a function for validating the password and storing the user name and valid password. There should be a function in main that creates an instance of the password creation object and enables/disables buttons as appropriate.

**Design/development documentation** – update the document to include screen captures of the functionality with explanations. Include all program code at the end of the document.

### Project Milestone #2

When account creation is successful, the Account Creation dialog should be destroyed or change to allow the user to login. Implement the login functionality by verifying that the account exists. The “Login” button should be disabled after it is clicked. The entry control for the password will use the <ENTER> key to accept the password.



## Appendix C

Hint: The call to the login function may not be able to be handled directly through the “command=” option of the button. A call to a function within the main GUI (outside the main loop) may need to be called which then calls the login function in the separate module.

Once a login has been successful, the main window should change to an interface for making a selection for the data access. This can be done by modifying the window or creating a new one.

Hint: To change the GUI, use the config and hide attributes, and the destroy function as appropriate to modify the window.

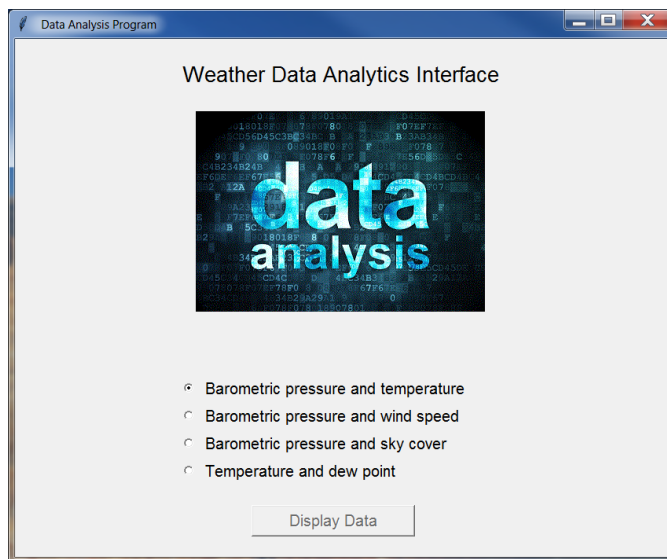
Example: `self.select_button.destroy()` # destroy the select button

Add the code to the login function to accept the user name and password when the <Enter> key is pressed by binding the entry control to a function call that tests for a match with the stored account information by calling a “verify” function. The entry control should accept input right aligned in the text box. If data entered and data password don’t match, red text should alert the user and the text box should be cleared - `entry.delete(0, END)`.

Note: The main window can morph (change) when the login is successful. There is no reason to create another dialog for this functionality.

Design the main GUI including the controls (functionality of the controls is not required at this point). Consider how the program will handle various situations, and how will the user interact with the program.

In preparation for demonstration of program operation, clean up the appearance of the program including the various displays and window handling. Windows should appear centered on the desktop, and not hide one another incorrectly. Controls should be aligned or centered with text explaining their functionality.



**Design/development documentation:** Update the design document screen captures and explanations, and include the updated code at the back of the document.



## Appendix C

### Project Milestone #3

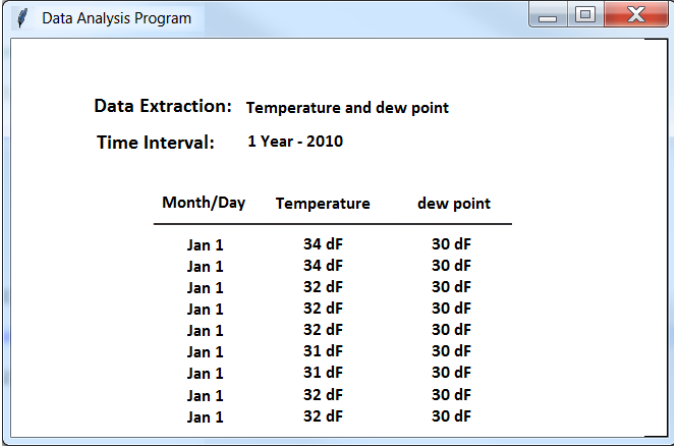
Complete the design for the main interface and implement the functionality for user selection of the data sets and time interval. Begin the data handling algorithm to read the file and extract the correct values. Consider creating a small file of data from the data set for testing. Determine the container type that will be used to store the data in the program, and how it will be extracted.

For data selection, radio buttons, option lists, and drop down menus are mutually exclusive and are possible solutions. The user should not type any information.

**Design/development documentation:** Update the design document screen captures, explanations, and paste the updated code in the back of the document.

### Project Milestone #4

Design the algorithm for extracting one set of data, and implement the window and output for displaying the columned data. Testing this functionality using a small data set instead of handling the entire file is recommended.



Data Extraction: Temperature and dew point		
Time Interval: 1 Year - 2010		
Month/Day	Temperature	dew point
Jan 1	34 dF	30 dF
Jan 1	34 dF	30 dF
Jan 1	32 dF	30 dF
Jan 1	32 dF	30 dF
Jan 1	32 dF	30 dF
Jan 1	31 dF	30 dF
Jan 1	31 dF	30 dF
Jan 1	32 dF	30 dF
Jan 1	32 dF	30 dF

Since the data consists of multiple samplings per day, consider how the user would prefer this to be handled. Also, consider how the plot functionality will be designed which may have an impact on the design at this point.

### Project Milestone #5

Design and implement the algorithms for selecting and displaying the other data sets and time intervals. Consider how this can be designed to accommodate any selection. There are a variety of ways to design this functionality.

Begin to consider the design for plotting the data selection.

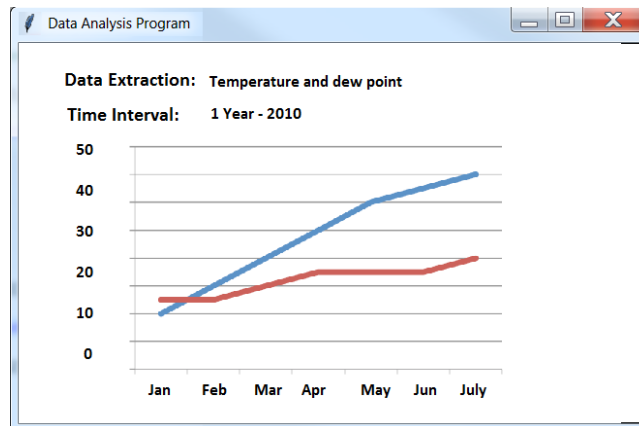
**Design/development documentation:** Update the design document explanations and screen captures, and include the updated code at the back of the document.

## Appendix C

### Project Milestone #6

Implement the plotting functionality and display. Test the program for all data sets and time intervals. Consider how the plot should be presented and how the axis should be labeled.

**Final preparation:** Test all areas of the program in preparation for demonstrating the working program. Update the design documentation with screen captures and updated code at the end, and prepare the final documentation package for submission.



### Final Presentation/Submission

Present the Design Document with program code, and demonstrate the running program operation. Submit the Design document including all code at the end as a Word file or pdf.

## Appendix D

### Links to Helpful Information

Matplotlib:

<https://matplotlib.org/>

Matplotlib Tutorial:

<https://matplotlib.org/3.1.1/tutorials/introductory/pyplot.html>

PEP 8 Style Guide for Python Code:

<https://www.python.org/dev/peps/pep-0008/>

Python Organization: Downloads, Documentation, etc.

<https://www.python.org/>

Python Tutorial:

<https://docs.python.org/3/tutorial/index.html>

## Appendix D

## Index

### A

acos(x) function,	13
active (state)	43
Agile Development	3
Addition (+) operator	
defined	12
concatenation	74
and operator	23
short circuit evaluation	24
Animation	125
appending data	
files (mode)	49
append function,	77
Arguments, passing	28
asin(x) function,	13
Assignment operator	10
atan(x) function,	13
axis labels	118

### B

bar chart	114
bool data type	25
Boolean	
expressions	23
logic	24
return values	31
Browser, launch	124
Buttons	
create	44
groups	92
widget	36
options	43
text	44

### C

Calendar	99
callback function	44
calling functions	27
Canvas component	113
case-sensitive	11
Centering windows	68
characters	73
comparing	24
copying	74
escape	11
finding in strings	75
indexing	74
newline	52
tab	52
stripping	52
Charts	
bar	114
flow	6
line	116
line example	117
pie	116
checkboxbutton control	36
Classes, example	103
Comments	9
config	80
Concatenation	74
lists	77
Conditional statements	23
Controlled loops	25
Copying	
characters	73
Count-controlled loops	26

---

## Index

### D

data	
appending to files	49
file design	69
reading from files	49
handling	69
writing to files	49
Data dictionary	69
data types	
bool	25
int	11
float	11
str	11
Decision structures	23
def (define)	27
degrees() function,	13
destroy()	80
Development	
Agile	3
cycle	4
methodologies	4
process	3
Dialog boxes	
error	21
designed	59
Information box	66
File Open	122
File, Save As	123
Message box	66
show (error, info, warning)	66
disabled (state)	43
Division	12
Drop-down menus	87

### E

e variable	13
else clause	23
else with try/except	57
endswith method	76
Entry control	
Text entry	65
focus	65
destroy()	80
example	65
Errors	
cost by phase	5
dialogs	66
IndexError	74
IOError	57
object	58
Traceback	20
SyntaxError	21
TypeError	52
ValueError	57
Escape characters	11
Event handler	
main loop	35
Exceptions	56
handling	57
exponentiation	12
example	13

### F

File modes	49
File objects	
close method	50

---



## Index

out of range	74	log() function	13
strings	73	Logical operators	24
Info dialog box	66	Loops	25
Initializer method	38	lower() method	52
int()	12	lstrip() method	52
IOError exception	57		
IPO document	3	<b>M</b>	
isalnum() method	76	Main function	28
isalpha() method	76	Main loop	37
isdigit() method	76	Mathematical operators	12
islower() method	76	matplotlib	114
isupper() method	76	animation	125
		module	115
		plotting	116
<b>J</b>		max function	77
Java	2	menu button	36
		Menus	87
		Methods, defined	15
<b>K</b>		Mixed-type expressions	13
keyboard input	12	min function	77
key words	11	minsize()	38
keyword arguments	30	Modularization	61
		Modules	115
		Modulus (%) operator	12
		Multiplication (*) operator	12
<b>L</b>		Mutable	76
Label control	36		
example	38	<b>N</b>	
Lambda	93	Negative indexes	73
len function	20	Newline (\n) character	
Line graph	117	adding	78
listbox	36	defined	52
list() function	78	removing	52
list to tuple	78		

---



## Index

Not in operator	75	pi variable	13
not operator	24	pie chart	116
Numbers		pip installer	35
floating point	11	plot() function	117
formatting	10	Pointers	32
integer	12	Precedence	13
random	14	print function	9
Numeric Lists	76	Program design	5
numpy module	125	Pseudocode	5
		Pyplot	115
<b>O</b>		<b>Q</b>	
Objects	32	Quit button	44
exception	58		
file	50		
StringVar	81		
Object Oriented	36		
open function	49		
Open file dialog	121		
Operators			
IN and NOT IN	76		
logical	24		
mathematical	12		
precedence of	13		
relational	24		
Option Lists control	84		
Output	2		
displaying	9		
file	53		
window	111		
<b>P</b>		<b>R</b>	
Parameter	30	radians() function	13
Passing arguments	29	Radio buttons	95
		random numbers	14
		range function	26
		Read, file	49
		Relational operators	22
		remove characters	52
		remove method	77
		replace method	76
		resizable	47
		return statements	33
		reverse method	77
		rounding	13
		rstrip() method	52
		<b>S</b>	
		“Save As” dialog	123

---

## Index

Saving programs	20	Truncation	12
search method	76	try/except	56
seeds, random number	14	Tuple	78
set method	81		
Showerror	66	<b>U</b>	
Showinfo()	66	UML Unified Modeling Language	3
Showwarning()	66	upper() method	52
sin()	13	User interface	2
Slicing	75		
sort() function	77	<b>V</b>	
split method	51	Variables	11
sqrt	13		
state	43	<b>W</b>	
startswith method	73	Web browser, launch	124
str type	11	Weight	47
strings	71	While loops	25
String lists	77	Wildcard import	45
String testing	76	Window	
StringVar object	81	border title	38
strip() method	52	centering	68
Subtraction (-) operator	12	chart	117
sys.exit()	68	destroy	80
		topmost	68
<b>T</b>		hide	121
tan() function	13	IDLE edit	19
Tearoff	88	interaction	103
Text on canvas	113	login	66
Text files	49	menu	87
tick marks	118	minsize()	38
Time	97	multiple	59
tkinter module	36	parent	105
tkinter main loop	37	writelines method	78
trace	91		
Traceback	20		

---

## Index

### X

x axis	117
x cords	117
XML	115

### Y

y axis	117
y cords	117

### Z

Zooming, plots	117
----------------	-----

---

