Functions

4

As the programs that we write grow, we need to take steps to make them easier to develop and debug. One way that we can do this is by breaking the program's code into sections called *functions*.

Functions serve several important purposes: They let us write code once and then call it from many locations, they allow us to test different parts of our solution individually, and they allow us to hide (or at least set aside) the details once we have completed part of our program. Functions achieve these goals by allowing the programmer to name and set aside a collection of Python statements for later use. Then our program can cause those statements to execute whenever they are needed. The statements are named by *defining* a function. The statements are executed by *calling* a function. When the statements in a function finish executing, control *returns* to the location where the function was called and the program continues to execute from that location.

The programs that you have written previously called functions like `print`, `input`, `int` and `float`. All of these functions have already been defined by the people that created the Python programming language, and these functions can be called in any Python program. In this chapter you will learn how to define and call your own functions, in addition to calling those that have been defined previously.

A function definition begins with a line that consists of `def`, followed by the name of the function that is being defined, followed by an open parenthesis, a close parenthesis and a colon. This line is followed by the body of the function, which is the collection of statements that will execute when the function is called. As with the bodies of `if` statements and loops, the bodies of functions are indented. A function's body ends before the next line that is indented the same amount as (or less than) the line that begins with `def`. For example, the following lines of code define a function that draws a box constructed from asterisk characters.

B. Stephenson, *The Python Workbook*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-18873-3_4

```
def drawBox():
  print("**********")
  print("*        *")
  print("*        *")
  print("**********")
```

On their own, these lines of code do not produce any output because, while the `drawBox` function has been defined, it is never called. Defining the function sets these statements aside for future use and associates the name `drawBox` with them, but it does not execute them. A Python program that consists of only these lines is a valid program, but it will not generate any output when it is executed.

The `drawBox` function is called by using its name, followed by an open parenthesis and a close parenthesis. Adding the following line to the end of the previous program (without indenting it) will call the function and cause the box to be drawn.

```
drawBox()
```

Adding a second copy of this line will cause a second box to be drawn and adding a third copy of it will cause a third box to be drawn. More generally, a function can be called as many times as needed when solving a problem, and those calls can be made from many different locations within the program. The statements in the body of the function execute every time the function is called. When the function returns execution continues with the statement immediately after the function call.

4.1 Functions with Parameters

The `drawBox` function works correctly. It draws the particular box that it was intended to draw, but it is not flexible, and as a result, it is not as useful as it could be. In particular, our function would be more flexible and useful if it could draw boxes of many different sizes.

Many functions take *arguments* which are values provided inside the parentheses when the function is called. The function receives these argument values in *parameter variables* that are included inside the parentheses when the function is defined. The number of parameter variables in a function's definition indicates the number of arguments that must be supplied when the function is called.

We can make the `drawBox` function more useful by adding two parameters to its definition. These parameters, which are separated by a comma, will hold the width of the box and the height of the box respectively. The body of the function uses the values in the parameter variables to draw the box, as shown below. An `if` statement and the `quit` function are used to end the program immediately if the arguments provided to the function are invalid.

```
## Draw a box outlined with asterisks and filled with spaces.
#  @param width the width of the box
#  @param height the height of the box
def drawBox(width, height):
  # A box that is smaller than 2x2 cannot be drawn by this function
  if width < 2 or height < 2:
    print("Error: The width or height is too small.")
    quit()

  # Draw the top of the box
  print("*" * width)

  # Draw the sides of the box
  for i in range(height - 2):
    print("*" + " " * (width - 2) + "*")

  # Draw the bottom of the box
  print("*" * width)
```

Two arguments must be supplied when the `drawBox` function is called because its definition includes two parameter variables. When the function executes the value of the first argument will be placed in the first parameter variable, and similarly, the value of the second argument will be placed in the second parameter variable. For example, the following function call draws a box with a width of 15 characters and a height of 4 characters. Additional boxes can be drawn with different sizes by calling the function again with different arguments.

```
drawBox(15, 4)
```

In its current form the `drawBox` function always draws the outline of the box with asterisk characters and it always fills the box with spaces. While this may work well in many circumstances there could also be times when the programmer needs a box drawn or filled with different characters. To accommodate this, we are going to update `drawBox` so that it takes two additional parameters which specify the outline and fill characters respectively. The body of the function must also be updated to use these additional parameter variables, as shown below. A call to the `drawBox` function which outlines the box with at symbols and fills the box with periods is included at the end of the program.

```
## Draw a box.
#  @param width the width of the box
#  @param height the height of the box
#  @param outline the character used for the outline of the box
#  @param fill the character used to fill the box
def drawBox(width, height, outline, fill):
  # A box that is smaller than 2x2 cannot be drawn by this function
  if width < 2 or height < 2:
    print("Error: The width or height is too small.")
    quit()

  # Draw the top of the box
  print(outline * width)

  # Draw the sides of the box
  for i in range(height - 2):
    print(outline + fill * (width - 2) + outline)
```

```
    # Draw the bottom of the box
    print(outline * width)

# Demonstrate the drawBox function
drawBox(14, 5, "@", ".")
```

The programmer will have to include the outline and fill values (in addition to the width and height) every time this version of `drawBox` is called. While needing to do so might be fine in some circumstances, it will be frustrating when asterisk and space are used much more frequently than other character combinations because these arguments will have to be repeated every time the function is called. To overcome this, we will add default values for the outline and fill parameters to the function's definition. The default value for a parameter is separated from its name by an equal sign, as shown below.

```
def drawBox(width, height, outline="*", fill=" "):
```

Once this change is made `drawBox` can be called with two, three or four arguments. If `drawBox` is called with two arguments, the first argument will be placed in the `width` parameter variable and the second argument will be placed in the `height` parameter variable. The `outline` and `fill` parameter variables will hold their default values of asterisk and space respectively. These default values are used because no arguments were provided for these parameters when the function was called.

Now consider the following call to drawBox:

```
drawBox(14, 5, "@", ".")
```

This function call includes four arguments. The first two arguments are the width and height, and they are placed into those parameter variables. The third argument is the outline character. Because it has been provided, the default outline value (asterisk) is replaced with the provided value, which is an at symbol. Similarly, because the call includes a fourth argument, the default fill value is replaced with a period. The box that results from the preceding call to `drawBox` is shown below.

```
@@@@@@@@@@@@@@
@............@
@............@
@............@
@@@@@@@@@@@@@@
```

4.2 Variables in Functions

When a variable is created inside a function the variable is *local* to that function. This means that the variable only exists when the function is executing and that it can only be accessed within the body of that function. The variable ceases to exist when the function returns, and as such, it cannot be accessed after that time. The `drawBox`

function uses several variables to perform its task. These include parameter variables such as `width` and `fill` that are created when the function is called, as well as the `for` loop control variable, `i`, that is created when the loop begins to execute. All of these are local variables that can only be accessed within this function. Variables created with assignment statements in the body of a function are also local variables.

4.3 Return Values

Our box-drawing function prints characters on the screen. While it takes arguments that specify how the box will be drawn, the function does not compute a result that needs to be stored in a variable and used later in the program. But many functions do compute such a value. For example, the `sqrt` function in the `math` module computes the square root of its argument and returns this value so that it can be used in subsequent calculations. Similarly, the `input` function reads a value typed by the user and then returns it so that it can be used later in the program. Some of the functions that you write will also need to return values.

A function returns a value using the `return` keyword, followed by the value that will be returned. When the `return` executes the function ends immediately and control returns to the location where the function was called. For example, the following statement immediately ends the function's execution and returns 5 to the location from which it was called.

```
return 5
```

Functions that return values are often called on the right side of an assignment statement, but they can also be called in other contexts where a value is needed. Examples of such include an `if` statement or `while` loop condition, or as an argument to another function, such as `print` or `range`.

A function that does not return a result does not need to use the `return` keyword because the function will automatically return after the last statement in the function's body executes. However, a programmer can use the `return` keyword, without a trailing value, to force the function to return at an earlier point in its body. Any function, whether it returns a value or not, can include multiple return statements. Such a function will return as soon as any of the return statements execute.

Consider the following example. A geometric sequence is a sequence of terms that begins with some value, a, followed by an infinite number of additional terms. Each term in the sequence, beyond the first, is computed by multiplying its immediate predecessor by r, which is referred to as the common ratio. As a result, the terms in the sequence are a, ar, ar^2, ar^3, ... When r is 1, the sum of the first n terms of a geometric sequence is $a \times n$. When r is not 1, the sum of the first n terms of a geometric sequence can be computed using the following formula.

$$\text{sum} = \frac{a(1 - r^n)}{1 - r}$$

A function can be written that computes the sum of the first n terms of any geometric sequence. It will require 3 parameters: `a`, `r` and `n`, and it will need to return one result, which is the sum of the first `n` terms. The code for the function is shown below.

```
## Compute the sum of the first n terms of a geometric sequence.
#  @param a the first term in the sequence
#  @param r the common ratio for the sequence
#  @param n the number of terms to include in the sum
#  @return the sum of the first n term of the sequence
def sumGeometric(a, r, n):
  # Compute and return the sum when the common ratio is 1
  if r == 1:
    return a * n

  # Compute and return the sum when the common ratio is not 1
  s = a * (1 - r ** n) / (1 - r)

  return s
```

The function begins by using an `if` statement to determine whether or not `r` is one. If it is, the sum is computed as `a * n` and the function immediately returns this value without executing the remaining lines in the function's body. When `r` is not equal to one, the body of the `if` statement is skipped and the sum of the first `n` terms is computed and stored in `s`. Then the value stored in `s` is returned to the location from which the function was called.

The following program demonstrates the `sumGeometric` function by computing sums until the user enters zero for `a`. Each sum is computed inside the function and then returned to the location where the function was called. Then the returned value is stored in the `total` variable using an assignment statement. A subsequent statement displays `total` before the program goes on and reads the values for another sequence from the user.

```
def main():
  # Read the initial value for the first sequence
  init = float(input("Enter the value of a (0 to quit): "))

  # While the initial value is non-zero
  while init != 0:
    # Read the ratio and number of terms
    ratio = float(input("Enter the ratio, r: "))
    num = int(input("Enter the number of terms, n: "))

    # Compute and display the total
    total = sumGeometric(init, ratio, num)
    print("The sum of the first", num, "terms is", total)

    # Read the initial value for the next sequence
    init = float(input("Enter the value of a (0 to quit): "))

# Call the main function
main()
```

4.4 Importing Functions into Other Programs

One of the benefits of using functions is the ability to write a function once and then call it many times from different locations. This is easily accomplished when the function definition and call locations all reside in the same file. The function is defined and then it is called by using its name, followed by parentheses containing any arguments.

At some point you will find yourself in the situation where you want to call a function that you wrote for a previous program while solving a new problem. New programmers (and even some experienced programmers) are often tempted to copy the function from the file containing the old program into the file containing the new one, but this is an undesirable approach. Copying the function results in the same code residing in two places. As a result, when a bug is identified it will need to be corrected twice. A better approach is to import the function from the old program into the new one, similar to the way that functions are imported from Python's built-in modules.

Functions from an old Python program can be imported into a new one using the `import` keyword, followed by the name of the Python file that contains the functions of interest (without the .py extension). This allows the new program to call all of the functions in the old file, but it also causes the program in the old file to execute. While this may be desirable in some situations, we often want access to the old program's functions without actually running the program. This is normally accomplished by creating a function named `main` that contains the statements needed to solve the problem. Then one line of code at the end of the file calls the `main` function. Finally, an `if` statement is added to ensure that the `main` function does not execute when the file has been imported into another program, as shown below:

```
if __name__ == "__main__":
  main()
```

This structure should be used whenever you create a program that includes functions that you might want to import into another program in the future.

4.5 Exercises

Functions allow us to name sequences of Python statements and call them from multiple locations within our program. This provides several advantages compared to programs that do not define any functions including the ability to write code once and call it from several locations, and the opportunity to test different parts of our solution individually. Functions also allow a programmer to set aside some of the program's details while concentrating on other aspects of the solution. Using functions effectively will help you write better programs, especially as you take on larger problems. Functions should be used when completing all of the exercises in this chapter.

Exercise 85: Compute the Hypotenuse

(*23 Lines*)

Write a function that takes the lengths of the two shorter sides of a right triangle as its parameters. Return the hypotenuse of the triangle, computed using Pythagorean theorem, as the function's result. Include a main program that reads the lengths of the shorter sides of a right triangle from the user, uses your function to compute the length of the hypotenuse, and displays the result.

Exercise 86: Taxi Fare

(*22 Lines*)

In a particular jurisdiction, taxi fares consist of a base fare of $4.00, plus $0.25 for every 140 meters travelled. Write a function that takes the distance travelled (in kilometers) as its only parameter and returns the total fare as its only result. Write a main program that demonstrates the function.

> Hint: Taxi fares change over time. Use constants to represent the base fare and the variable portion of the fare so that the program can be updated easily when the rates increase.

Exercise 87: Shipping Calculator

(*23 Lines*)

An online retailer provides express shipping for many of its items at a rate of $10.95 for the first item in an order, and $2.95 for each subsequent item in the same order. Write a function that takes the number of items in the order as its only parameter. Return the shipping charge for the order as the function's result. Include a main program that reads the number of items purchased from the user and displays the shipping charge.

Exercise 88: Median of Three Values

(*Solved, 43 Lines*)

Write a function that takes three numbers as parameters, and returns the median value of those parameters as its result. Include a main program that reads three values from the user and displays their median.

> Hint: The median value is the middle of the three values when they are sorted into ascending order. It can be found using if statements, or with a little bit of mathematical creativity.

Exercise 89: Convert an Integer to Its Ordinal Number

(*47 Lines*)

Words like *first*, *second* and *third* are referred to as ordinal numbers. In this exercise, you will write a function that takes an integer as its only parameter and returns a string containing the appropriate English ordinal number as its only result. Your function must handle the integers between 1 and 12 (inclusive). It should return an empty string if the function is called with an argument outside of this range. Include a main program that demonstrates your function by displaying each integer from 1 to 12 and its ordinal number. Your main program should only run when your file has not been imported into another program.

Exercise 90: The Twelve Days of Christmas

(*Solved, 52 Lines*)

The Twelve Days of Christmas is a repetitive song that describes an increasingly long list of gifts sent to one's true love on each of 12 days. A single gift is sent on the first day. A new gift is added to the collection on each additional day, and then the complete collection is sent. The first three verses of the song are shown below. The complete lyrics are available on the Internet.

On the first day of Christmas
my true love sent to me:
A partridge in a pear tree.

On the second day of Christmas
my true love sent to me:
Two turtle doves,
And a partridge in a pear tree.

On the third day of Christmas
my true love sent to me:
Three French hens,
Two turtle doves,
And a partridge in a pear tree.

Write a program that displays the complete lyrics for The Twelve Days of Christmas. Your program should include a function that displays one verse of the song. It will take the verse number as its only parameter. Then your program should call this function 12 times with integers that increase from 1 to 12.

Each item that is sent to the recipient in the song should only appear in your program once, with the possible exception of the partridge. It may appear twice if that helps you handle the difference between "A partridge in a pear tree" in the first verse and "And a partridge in a pear tree" in the subsequent verses. Import your solution to Exercise 89 to help you complete this exercise.

Exercise 91: Gregorian Date to Ordinal Date

(*72 Lines*)

An ordinal date consists of a year and a day within it, both of which are integers. The year can be any year in the Gregorian calendar while the day within the year ranges from one, which represents January 1, through to 365 (or 366 if the year is a leap year) which represents December 31. Ordinal dates are convenient when computing differences between dates that are a specific number of days (rather than months). For example, ordinal dates can be used to easily determine whether a customer is within a 90 day return period, the sell-by date for a food-product based on its production date, and the due date for a baby.

Write a function named `ordinalDate` that takes three integers as parameters. These parameters will be a day, month and year respectively. The function should return the day within the year for that date as its only result. Create a main program that reads a day, month and year from the user and displays the day within the year for that date. Your main program should only run when your file has not been imported into another program.

Exercise 92: Ordinal Date to Gregorian Date

(*103 Lines*)

Create a function that takes an ordinal date, consisting of a year and a day within in that year, as its parameters. The function will return the day and month corresponding to that ordinal date as its results. Ensure that your function handles leap years correctly.

Use your function, as well as the `ordinalDate` function that you wrote previously, to create a program that reads a date from the user. Then your program should report a second date that occurs some number of days later. For example, your program could read the date a product was purchased and then report the last date that it can be returned (based on a return period that is a particular number of days), or your program could compute the due date for a baby based on a gestation period of 280 days. Ensure that your program correctly handles cases where the entered date and the computed date occur in different years.

Exercise 93: Center a String in the Terminal Window

(*Solved, 29 Lines*)

Write a function that takes a string, *s*, as its first parameter, and the width of the window in characters, *w*, as its second parameter. Your function will return a new string that includes whatever leading spaces are needed so that *s* will be centered in the window when the new string is printed. The new string should be constructed in the following manner:

- If the length of *s* is greater than or equal to the width of the window then *s* should be returned.

- If the length of *s* is less than the width of the window then a string containing `(len(s) - w) // 2` spaces followed by *s* should be returned.

Write a main program that demonstrates your function by displaying multiple strings centered in the window.

Exercise 94: Is It a Valid Triangle?

(*33 Lines*)

If you have 3 straws, possibly of differing lengths, it may or may not be possible to lay them down so that they form a triangle when their ends are touching. For example, if all of the straws have a length of 6 inches then one can easily construct an equilateral triangle using them. However, if one straw is 6 inches long, while the other two are each only 2 inches long, then a triangle cannot be formed. More generally, if any one length is greater than or equal to the sum of the other two then the lengths cannot be used to form a triangle. Otherwise they can form a triangle.

Write a function that determines whether or not three lengths can form a triangle. The function will take 3 parameters and return a Boolean result. If any of the lengths are less than or equal to 0 then your function should return `False`. Otherwise it should determine whether or not the lengths can be used to form a triangle using the method described in the previous paragraph, and return the appropriate result. In addition, write a program that reads 3 lengths from the user and demonstrates the behaviour of your function.

Exercise 95: Capitalize It

(*Solved, 68 Lines*)

Many people do not use capital letters correctly, especially when typing on small devices like smart phones. To help address this situation, you will create a function that takes a string as its only parameter and returns a new copy of the string that has been correctly capitalized. In particular, your function must:

- Capitalize the first non-space character in the string,
- Capitalize the first non-space character after a period, exclamation mark or question mark, and
- Capitalize a lowercase "i" if it is preceded by a space and followed by a space, period, exclamation mark, question mark or apostrophe.

Implementing these transformations will correct most capitalization errors. For example, if the function is provided with the string "what time do i have to be there? what's the address? this time i'll try to be on time!" then it should return the string "What time do I have to be there? What's the address? This time I'll try to be on time!". Include a main program that reads a string from the user, capitalizes it using your function, and displays the result.

Exercise 96: Does a String Represent an Integer?

(*Solved, 30 Lines*)

In this exercise you will write a function named `isInteger` that determines whether or not the characters in a string represent a valid integer. When determining if a string represents an integer you should ignore any leading or trailing white space. Once this white space is ignored, a string represents an integer if its length is at least one and it only contains digits, or if its first character is either + or – and the first character is followed by one or more characters, all of which are digits.

Write a main program that reads a string from the user and reports whether or not it represents an integer. Ensure that the main program will not run if the file containing your solution is imported into another program.

> Hint: You may find the `lstrip`, `rstrip` and/or `strip` methods for strings helpful when completing this exercise. Documentation for these methods is available online.

Exercise 97: Operator Precedence

(*30 Lines*)

Write a function named `precedence` that returns an integer representing the precedence of a mathematical operator. A string containing the operator will be passed to the function as its only parameter. Your function should return 1 for + and –, 2 for * and /, and 3 for ^. If the string passed to the function is not one of these operators then the function should return –1. Include a main program that reads an operator from the user and either displays the operator's precedence or an error message indicating that the input was not an operator. Your main program should only run when the file containing your solution has not been imported into another program.

> In this exercise, along with others that appear later in the book, we will use ^ to represent exponentiation. Using ^ instead of Python's choice of ** will make these exercises easier because an operator will always be a single character.

Exercise 98: Is a Number Prime?

(*Solved, 28 Lines*)

A prime number is an integer greater than one that is only divisible by one and itself. Write a function that determines whether or not its parameter is prime, returning `True` if it is, and `False` otherwise. Write a main program that reads an integer from the user and displays a message indicating whether or not it is prime. Ensure that the main program will not run if the file containing your solution is imported into another program.

Exercise 99: Next Prime

(*27 Lines*)

In this exercise you will create a function named `nextPrime` that finds and returns the first prime number larger than some integer, *n*. The value of *n* will be passed to the function as its only parameter. Include a main program that reads an integer from the user and displays the first prime number larger than the entered value. Import and use your solution to Exercise 98 while completing this exercise.

Exercise 100: Random Password

(*Solved, 33 Lines*)

Write a function that generates a random password. The password should have a random length of between 7 and 10 characters. Each character should be randomly selected from positions 33 to 126 in the ASCII table. Your function will not take any parameters. It will return the randomly generated password as its only result. Display the randomly generated password in your file's main program. Your main program should only run when your solution has not been imported into another file.

> Hint: You will probably find the `chr` function helpful when completing this exercise. Detailed information about this function is available online.

Exercise 101: Random License Plate

(*45 Lines*)

In a particular jurisdiction, older license plates consist of three letters followed by three digits. When all of the license plates following that pattern had been used, the format was changed to four digits followed by three letters.

Write a function that generates a random license plate. Your function should have approximately equal odds of generating a sequence of characters for an old license plate or a new license plate. Write a main program that calls your function and displays the randomly generated license plate.

Exercise 102: Check a Password

(*Solved, 40 Lines*)

In this exercise you will write a function that determines whether or not a password is good. We will define a good password to be a one that is at least 8 characters long and contains at least one uppercase letter, at least one lowercase letter, and at least one number. Your function should return `True` if the password passed to it as its only parameter is good. Otherwise it should return `False`. Include a main program that reads a password from the user and reports whether or not it is good. Ensure that your main program only runs when your solution has not been imported into another file.

Exercise 103: Random Good Password

(*22 Lines*)

Using your solutions to Exercises 100 and 102, write a program that generates a random good password and displays it. Count and display the number of attempts that were needed before a good password was generated. Structure your solution so that it imports the functions you wrote previously and then calls them from a function named `main` in the file that you create for this exercise.

Exercise 104: Hexadecimal and Decimal Digits

(*41 Lines*)

Write two functions, `hex2int` and `int2hex`, that convert between hexadecimal digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E and F) and decimal (base 10) integers. The `hex2int` function is responsible for converting a string containing a single hexadecimal digit to a base 10 integer, while the `int2hex` function is responsible for converting an integer between 0 and 15 to a single hexadecimal digit. Each function will take the value to convert as its only parameter and return the converted value as its only result. Ensure that the `hex2int` function works correctly for both uppercase and lowercase letters. Your functions should display a meaningful error message and end the program if the parameter's value is outside of the expected range.

Exercise 105: Arbitrary Base Conversions

(*Solved, 71 Lines*)

Write a program that allows the user to convert a number from one base to another. Your program should support bases between 2 and 16 for both the input number and the result number. If the user chooses a base outside of this range then an appropriate error message should be displayed and the program should exit. Divide your program into several functions, including a function that converts from an arbitrary base to base 10, a function that converts from base 10 to an arbitrary base, and a main program that reads the bases and input number from the user. You may find your solutions to Exercises 81, 82 and 104 helpful when completing this exercise.

Exercise 106: Days in a Month

(*47 Lines*)

Write a function that determines how many days there are in a particular month. Your function will take two parameters: The month as an integer between 1 and 12, and the year as a four digit integer. Ensure that your function reports the correct number of days in February for leap years. Include a main program that reads a month and year from the user and displays the number of days in that month. You may find your solution to Exercise 58 helpful when solving this problem.

Exercise 107: Reduce a Fraction to Lowest Terms

(*Solved, 46 Lines*)

Write a function that takes two positive integers that represent the numerator and denominator of a fraction as its only parameters. The body of the function should reduce the fraction to lowest terms and then return both the numerator and denominator of the reduced fraction as its result. For example, if the parameters passed to the function are 6 and 63 then the function should return 2 and 21. Include a main program that allows the user to enter a numerator and denominator. Then your program should display the reduced fraction.

> Hint: In Exercise 79 you wrote a program for computing the greatest common divisor of two positive integers. You may find that code useful when completing this exercise.

Exercise 108: Reduce Measures

(*Solved, 87 Lines*)

Many recipe books still use cups, tablespoons and teaspoons to describe the volumes of ingredients used when cooking or baking. While such recipes are easy enough to follow if you have the appropriate measuring cups and spoons, they can be difficult to double, triple or quadruple when cooking Christmas dinner for the entire extended family. For example, a recipe that calls for 4 tablespoons of an ingredient requires 16 tablespoons when quadrupled. However, 16 tablespoons would be better expressed (and easier to measure) as 1 cup.

Write a function that expresses an imperial volume using the largest units possible. The function will take the number of units as its first parameter, and the unit of measure (cup, tablespoon or teaspoon) as its second parameter. It will return a string representing the measure using the largest possible units as its only result. For example, if the function is provided with parameters representing 59 teaspoons then it should return the string "1 cup, 3 tablespoons, 2 teaspoons".

> Hint: One cup is equivalent to 16 tablespoons. One tablespoon is equivalent to 3 teaspoons.

Exercise 109: Magic Dates

(*Solved, 26 Lines*)

A magic date is a date where the day multiplied by the month is equal to the two digit year. For example, June 10, 1960 is a magic date because June is the sixth month, and 6 times 10 is 60, which is equal to the two digit year. Write a function that determines whether or not a date is a magic date. Use your function to create a main program that finds and displays all of the magic dates in the 20th century. You will probably find your solution to Exercise 106 helpful when completing this exercise.