Part II
Solutions

Introduction to Programming Solutions

9

Solution to Exercise 1: Mailing Address

```
##
# Display a person's complete mailing address.
#
print("Ben Stephenson")
print("Department of Computer Science")
print("University of Calgary")
print("2500 University Drive NW")
print("Calgary, Alberta T2N 1N4")
print("Canada")
```

Solution to Exercise 3: Area of a Room

```
##
# Compute the area of a room.
#

# Read the input values from the user
length = float(input("Enter the length of the room in feet: "))
width = float(input("Enter the width of the room in feet: "))

# Compute the area of the room
area = length * width

# Display the result
print("The area of the room is", area, "square feet")
```

The `float` function is used to convert the user's input into a number.

In Python, multiplication is performed using the $*$ operator.

B. Stephenson, *The Python Workbook*, DOI 10.1007/978-3-319-14240-1_9

Solution to Exercise 4: Area of a Field

```
##
#  Compute the area of a field, reporting the result in acres.
#
SQFT_PER_ACRE = 43560

# Read input from the user
length = float(input("Enter the length of the field in feet: "))
width = float(input("Enter the width of the field in feet: "))

# Compute the area in acres
acres = length * width / SQFT_PER_ACRE

# Display the result
print("The area of the field is", acres, "acres")
```

Solution to Exercise 5: Bottle Deposits

```
##
#  Compute the refund amount for a collection of bottles.
#
LESS_DEPOSIT = 0.10
MORE_DEPOSIT = 0.25

# Read input from the user
less = int(input("How many containers 1 litre or less do you have? "))
more = int(input("How many containers more than 1 litre do you have? "))

# Compute the refund amount
refund = less * LESS_DEPOSIT + more * MORE_DEPOSIT

# Display the result
print("Your total refund will be $%.2f." % refund)
```

The `%.2f` format specifier indicates that a value should be formatted as a floating point number with 2 digits to the right of the decimal point.

Solution to Exercise 6: Tax and Tip

```
##
#  Compute the tax and tip for a restaurant meal.
#
TAX_RATE = 0.05
TIP_RATE = 0.18

# Read the cost of the meal from the user
cost = float(input("Enter the cost of the meal: "))

# Compute the tax and the tip
tax = cost * TAX_RATE
tip = cost * TIP_RATE
total = cost + tax + tip

# Display the result
print("The tax is %.2f and the tip is %.2f, making the total %.2f" % \
      (tax, tip, total))
```

My local tax rate is 5%. In Python we represent 5% and 18% as 0.05 and 0.18 respectively.

The \ at the end of the line is called the line continuation character. It tells Python that the statement continues on the next line. Do not include any spaces or tabs after the \ character.

Solution to Exercise 7: Sum of the First *n* Positive Integers

```
##
#  Compute the sum of the first n positive integers.
#

# Read input from the user
n = int(input("Enter a positive integer: "))

# Compute the sum
sm = n * (n + 1) / 2

# Display the result
print("The sum of the first", n, "positive integers is", sm)
```

Python includes a built-in function named `sum`. As a result, we will use a different name for our variable.

Solution to Exercise 10: Arithmetic

```
##
#  Demonstrate Python's mathematical operators and its math module.
#
from math import log10

# Read input values from the user
a = int(input("Enter the value of a: "))
b = int(input("Enter the value of b: "))

# Compute and display the sum, difference, product,
# quotient and remainder
print(a, "+", b, "is", a + b)
print(a, "-", b, "is", a - b)
print(a, "*", b, "is", a * b)
print(a, "/", b, "is", a / b)
print(a, "%", b, "is", a % b)

# Compute the logarithm and the power
print("The base 10 logarithm of", a, "is", log10(a))
print(a, "^", b, "is", a**b)
```

We must import the `log10` function from the `math` module before we call it. Import statements normally appear at the top of the file.

The remainder is computed using the `%` operator.

Solution to Exercise 13: Making Change

```
##
#  Compute the minimum collection of coins needed to represent a number of cents.
#
CENTS_PER_TOONIE = 200
CENTS_PER_LOONIE = 100
CENTS_PER_QUARTER = 25
CENTS_PER_DIME =    10
CENTS_PER_NICKEL =   5

# Read the number of cents from the user
cents = int(input("Enter the number of cents: "))
```

```
# Determine the number of toonies by performing an integer division by 200.  Then compute
# the amount of change that still needs to be considered by
# computing the remainder after dividing by 200.
print(" ", cents // CENTS_PER_TOONIE, "toonies")
cents = cents % CENTS_PER_TOONIE

# Repeat the process for loonies, quarters, dimes, and nickels
print(" ", cents // CENTS_PER_LOONIE, "loonies")
cents = cents % CENTS_PER_LOONIE

print(" ", cents // CENTS_PER_QUARTER, "quarters")
cents = cents % CENTS_PER_QUARTER

print(" ", cents // CENTS_PER_DIME, "dimes")
cents = cents % CENTS_PER_DIME

print(" ", cents // CENTS_PER_NICKEL, "nickels")
cents = cents % CENTS_PER_NICKEL

# Display the number of pennies
print(" ", cents, "pennies")
```

Integer division, which discards any fractional part of the result, is performed using the // operator.

Solution to Exercise 14: Height Units

```
##
#  Convert a height in feet and inches to centimeters.
#
IN_PER_FT = 12
CM_PER_IN = 2.54

# Read input from the user
print("Enter your height:")
feet = int(input("  Number of feet: "))
inches = int(input("  Number of inches: "))

# Compute the equivalent number of centimeters
cm = (feet * IN_PER_FT + inches) * CM_PER_IN

# Display the result
print("Your height in centimeters is:", cm)
```

Solution to Exercise 17: Heat Capacity

```
##
#  Compute the amount of energy needed to heat a volume of water, and the cost of doing so.
#

# Define constants for the specific heat capacity of water and the price of electricity
WATER_HEAT_CAPACITY = 4.186
ELECTRICITY_PRICE = 8.9
J_TO_KWH = 2.777e-7
```

> Python allows numbers to be written in scientific notation by placing the coefficient to the left of an `e` and the exponent to its right. As a result, $2.777 * 10^{-7}$ is written as `2.777e-7`.

```
# Read the volume from the user
volume = float(input("Enter the amount of water in milliliters: "))
d_temp = float(input("Enter the temperature increase (degrees Celsius): "))

# Compute the energy in Joules
q = volume * d_temp * WATER_HEAT_CAPACITY

# Display the result in Joules
print("That will require %d Joules of energy." % q)

# Compute the cost
kwh = q * J_TO_KWH
cost = kwh * ELECTRICITY_PRICE

# Display the cost
print("That much energy will cost %.2f cents." % \
      cost)
```

> Because water has a density of 1 gram per milliliter grams and milliliters can be used interchangeably. Prompting the user for milliliters makes the program easier to use because most people think about the volume of water in a coffee cup, not its mass.

Solution to Exercise 19: Free Fall

```
##
#  Compute the speed of an object when it hits the ground after being dropped.
#
from math import sqrt

# Define a constant for the acceleration due to gravity in m/s**2
GRAVITY = 9.8

# Read the height from which the object is dropped
d = float(input("Height from which the object is dropped (in meters): "))
```

```
# Compute the final velocity
vf = sqrt(2 * GRAVITY * d)

# Display the result
print("It will hit the ground at %.2f m/s." % vf)
```

The v_i^2 term has not been included in the calculation because v_i is 0.

Solution to Exercise 23: Area of a Regular Polygon

```
##
#  Compute the area of a regular polygon.
#
from math import tan, pi

# Read input from the user
s = float(input("Enter the length of each side of the polygon: "))
n = int(input("Enter the number of sides: "))

# Compute the area of the polygon
area = (n * s ** 2) / (4 * tan(pi / n))

# Display the result
print("The area of the polygon is", area)
```

We have chosen to cast input `n` to an integer because a polygon cannot have a fractional number of sides.

Solution to Exercise 25: Units of Time (Again)

```
##
#  Convert a number of seconds to days, hours, minutes and seconds.
#
SECONDS_PER_DAY = 86400
SECONDS_PER_HOUR = 3600
SECONDS_PER_MINUTE = 60

# Read input from the user
seconds = int(input("Enter a number of seconds: "))

# Compute the days, hours, minutes and seconds
days = seconds / SECONDS_PER_DAY
seconds = seconds % SECONDS_PER_DAY

hours = seconds / SECONDS_PER_HOUR
seconds = seconds % SECONDS_PER_HOUR

minutes = seconds / SECONDS_PER_MINUTE
seconds = seconds % SECONDS_PER_MINUTE

# Display the result with the desired formatting
print("The equivalent duration is", \
      "%d:%02d:%02d:%02d." % (days, hours, minutes, seconds))
```

The `%02d` format specifier tells Python to format the integer using two digits, adding a leading 0 if necessary.

Solution to Exercise 28: Wind Chill

```
##
# Compute the wind chill index for a given air temperature and wind speed
#
WC_OFFSET = 13.12
WC_FACTOR1 = 0.6215
WC_FACTOR2 = -11.37
WC_FACTOR3 = 0.3965
WC_EXPONENT = 0.16
```

Computing wind chill requires several numeric constants that were determined by scientists and medical experts.

```
# Read the air temperature and wind speed from the user
temp = float(input("Enter the air temperature (degrees Celsius): "))
speed = float(input("Enter the wind speed (kilometers per hour): "))

# Compute the wind chill index
wci = WC_OFFSET + \
      WC_FACTOR1 * temp + \
      WC_FACTOR2 * speed ** WC_EXPONENT + \
      WC_FACTOR3 * temp * speed ** WC_EXPONENT

# Display the result rounded to the closest integer
print("The wind chill index is", round(wci))
```

Solution to Exercise 32: Sort 3 Integers

```
##
# Sort 3 values entered by the user into increasing order.
#

# Read the numbers from the user, naming them a, b and c
a = int(input("Enter the first number: "))
b = int(input("Enter the second number: "))
c = int(input("Enter the third number: "))
```

```
mn = min(a,b,c)              # the minimum value
mx = max(a,b,c)              # the maximum value
md = a + b + c - mn - mx     # the middle value

# Display the result
print("The numbers in sorted order are:")
print(" ", mn)
print(" ", md)
print(" ", mx)
```

Since `min` and `max` are the names of functions in Python we shouldn't use those names for variables. Instead we use variables named `mn` and `mx` to hold the minimum and maximum values respectively.

Solution to Exercise 33: Day Old Bread

```
##
# Compute the price of a day old bread order.
#
BREAD_PRICE = 3.49
DISCOUNT_RATE = 0.60

# Read the number of loaves from the user
num_loaves = int(input("Enter the number of day old loaves: "))

# Compute the discount and total price
regular_price = num_loaves * BREAD_PRICE
discount = regular_price * DISCOUNT_RATE
total = regular_price - discount

# Display the result
print("Regular price: %5.2f" % regular_price)
print("Discount:      %5.2f" % discount)
print("Total:         %5.2f" % total)
```

The `%5.2f` format tells Python that a total of at least 5 spaces should be used to display the number, with 2 digits to the right of the decimal point. This will help keep the columns lined up when the number of digits needed for the discount and the total are different.

If Statement Solutions

10

Solution to Exercise 34: Even or Odd?

```
##
#  Determine and display whether an integer entered by the user is even or odd.
#

# Read the integer from the user
num = int(input("Enter an integer: "))

# Determine whether it is even or odd by using the
# modulus (remainder) operator
if num % 2 == 1:
  print(num, "is odd.")
else:
  print(num, "is even.")
```

Dividing an even number by 2 always results in a remainder of 0. Dividing an odd number by 2 always results in a remainder of 1.

Solution to Exercise 36: Vowel or Consonant

```
##
#  Determine if a letter is a vowel or a consonant.
#

# Read a letter from the user
letter = input("Enter a letter of the alphabet: ")

# Classify the letter and report the result
if letter == "a" or letter == "e" or \
   letter == "i" or letter == "o" or \
   letter == "u":
  print("It's a vowel.")
elif letter == "y":
  print("Sometimes it's a vowel... Sometimes it's a consonant.")
else:
  print("It's a consonant.")
```

This version of the program only works for lowercase letters. You can add support for uppercase letters by including additional comparisons that follow the same pattern.

B. Stephenson, *The Python Workbook*, DOI 10.1007/978-3-319-14240-1_10

Solution to Exercise 37: Name that Shape

```
##
#  Report the name of a shape from its number of sides.
#

# Read the number of sides from the user
nsides = int(input("Enter the number of sides: "))

# Determine the name, leaving it empty if an unsupported number of sides was entered
name = ""
if nsides == 3:
  name = "triangle"
elif nsides == 4:
  name = "quadrilateral"
elif nsides == 5:
  name = "pentagon"
elif nsides == 6:
  name = "hexagon"
elif nsides == 7:
  name = "heptagon"
elif nsides == 8:
  name = "octagon"
elif nsides == 9:
  name = "nonagon"
elif nsides == 10:
  name = "decagon"

# Display an error message or the name of the polygon
if name == "":
  print("That number of sides is not supported by this program.")
else:
  print("That's a", name)
```

The empty string is being used as a sentinel value. If the number of sides entered by the user is outside of the supported range then `name` will remain empty, causing an error message to be displayed later in the program.

Solution to Exercise 38: Month Name to Number of Days

```
##
#  Display the number of days in a month.
#

# Read input from the user
month = input("Enter the name of a month: ")

# Compute the number of days in the month
days = 31
```

Start by assuming that the number of days is 31. Then update the number of days if necessary.

```
if month == "April" or month == "June" or \
   month == "September" or month == "November":
  days = 30
elif month == "February":
  days = "28 or 29"

# Display the result
print(month, "has", days, "days in it.")
```

When `month` is February, the value assigned to `days` is a string so that we can represent 28 or 29 days.

Solution to Exercise 40: Name that Triangle

```
##
# Determine the name of a triangle from the lengths of its sides.
#

# Read the side lengths from the user
side1 = float(input("Enter the length of side 1: "))
side2 = float(input("Enter the length of side 2: "))
side3 = float(input("Enter the length of side 3: "))

# Determine the triangle's name
if side1 == side2 and side2 == side3:
  name = "equilateral"
elif side1 == side2 or side2 == side3 or \
     side3 == side1:
  name = "isoceles"
else:
  name = "scalene"

# Display the triangle's name
print("That's a", name, "triangle")
```

We could also check that `side1` is equal to `side3` as part of the condition for an equilateral triangle. However, that comparison isn't necessary because the == operator is transitive.

Solution to Exercise 41: Note to Frequency

```
##
# Convert the name of a note to its frequency.
#
C4_FREQ = 261.63
D4_FREQ = 293.66
E4_FREQ = 329.63
F4_FREQ = 349.23
G4_FREQ = 392.00
A4_FREQ = 440.00
B4_FREQ = 493.88

# Read the note name from the user
name = input("Enter the two character note name, such as C4: ")

# Store the note and its octave in separate variables
note = name[0]
octave = int(name[1])
```

```
# Get the frequency of the note, assuming it is in the fourth octave
if note == "C":
  freq = C4_FREQ
elif note == "D":
  freq = D4_FREQ
elif note == "E":
  freq = E4_FREQ
elif note == "F":
  freq = F4_FREQ
elif note == "G":
  freq = G4_FREQ
elif note == "A":
  freq = A4_FREQ
elif note == "B":
  freq = B4_FREQ

# Now adjust the frequency to bring it into the correct octave
freq = freq / 2 ** (4 - octave)

# Display the result
print("The frequncy of", name, "is", freq)
```

Solution to Exercise 42: Frequency to Note

```
##
#  Read a frequency from the user and display the note (if any) that it corresponds to.
#
C4_FREQ = 261.63
D4_FREQ = 293.66
E4_FREQ = 329.63
F4_FREQ = 349.23
G4_FREQ = 392.00
A4_FREQ = 440.00
B4_FREQ = 493.88
LIMIT = 1

# Read the frequency from the user
freq = float(input("Enter a frequency: "))

# Determine the note that corresponds to the entered frequency.  Set
# note equal to the empty string if there isn't a match.
if freq >= C4_FREQ - LIMIT and freq <= C4_FREQ + LIMIT:
  note = "C4"
elif freq >= D4_FREQ - LIMIT and freq <= D4_FREQ + LIMIT:
  note = "D4"
elif freq >= E4_FREQ - LIMIT and freq <= E4_FREQ + LIMIT:
  note = "E4"
elif freq >= F4_FREQ - LIMIT and freq <= F4_FREQ + LIMIT:
  note = "F4"
elif freq >= G4_FREQ - LIMIT and freq <= G4_FREQ + LIMIT:
  note = "G4"
```

```
elif freq >= A4_FREQ - LIMIT and freq <= A4_FREQ + LIMIT:
  note = "A4"
elif freq >= B4_FREQ - LIMIT and freq <= B4_FREQ + LIMIT:
  note = "B4"
else:
  note = ""

# Display the result, or an appropriate error message
if note == "":
  print("There is no note that corresponds to that frequency.")
else:
  print("That frequency is", note)
```

Solution to Exercise 46: Season from Month and Day

```
##
# Determine and display the season associated with a date.
#

# Read the date from the user
month = input("Enter the name of the month: ")
day = int(input("Enter the day number: "))

# Determine the season
if month == "January" or month == "February":
  season = "Winter"
elif month == "March":
  if day < 20:
    season = "Winter"
  else:
    season = "Spring"
elif month == "April" or month == "May":
  season = "Spring"
elif month == "June":
  if day < 21:
    season = "Spring"
  else:
    season = "Summer"
elif month == "July" or month == "August":
  season = "Summer"
elif month == "September":
  if day < 22:
    season = "Summer"
  else:
    season = "Fall"
elif month == "October" or month == "November":
  season = "Fall"
elif month == "December":
  if day < 21:
    season = "Fall"
  else:
    season = "Winter"
```

This solution to the season problem uses several `elif` statements so that the conditions remain as simple as possible. Another way of approaching this problem is to minimize the number of `elif` statements by making the conditions more complex.

```
# Display the result
print(month, day, "is in", season)
```

Solution to Exercise 48: Chinese Zodiac

```
##
#  Determine the animal associated with a year according to the Chinese zodiac.
#

# Read a year from the user
year = int(input("Enter a year: "))

# Determine the animal associated with that year
if year % 12 == 8:
  animal = "Dragon"
elif year % 12 == 9:
  animal = "Snake"
elif year % 12 == 10:
  animal = "Horse"
elif year % 12 == 11:
  animal = "Sheep"
elif year % 12 == 0:
  animal = "Monkey"
elif year % 12 == 1:
  animal = "Rooster"
elif year % 12 == 2:
  animal = "Dog"
elif year % 12 == 3:
  animal = "Pig"
elif year % 12 == 4:
  animal = "Rat"
elif year % 12 == 5:
  animal = "Ox"
elif year % 12 == 6:
  animal = "Tiger"
elif year % 12 == 7:
  animal = "Hare"

# Report the result
print("%d is the year of the %s." % (year, animal))
```

When multiple items are formatted all of the values are placed inside parentheses on the right side of the `%` operator.

Solution to Exercise 51: Letter Grade to Grade Points

```
##
# Convert from a letter grade to a number of grade points.
#
A       = 4.0
A_MINUS = 3.7
B_PLUS  = 3.3
B       = 3.0
B_MINUS = 2.7
C_PLUS  = 2.3
C       = 2.0
C_MINUS = 1.7
D_PLUS  = 1.3
D       = 1.0
F       = 0
INVALID = -1

# Read the letter grade from the user
letter = input("Enter a letter grade: ")
letter = letter.upper()
```

The statement `letter = letter.upper()` converts any lowercase letters entered by the user into uppercase letters, storing the result back into the same variable. Including this statement allows the program to work with lowercase letters without including them in the conditions of the `if` and `elif` statements.

```
# Convert from a letter grade to a number of grade points using -1 grade points as a sentinel
# value indicating invalid input
if letter == "A+" or letter == "A":
  gp = A
elif letter == "A-":
  gp = A_MINUS
elif letter == "B+":
  gp = B_PLUS
elif letter == "B":
  gp = B
elif letter == "B-":
  gp = B_MINUS
elif letter == "C+":
  gp = C_PLUS
elif letter == "C":
  gp = C
elif letter == "C-":
  gp = C_MINUS
elif letter == "D+":
  gp = D_PLUS
```

```
elif letter == "D":
  gp = D
elif letter == "F":
  gp = F
else:
  gp = INVALID

# Display the output
if gp == INVALID:
  print("That wasn't a valid number of grade points.")
else:
  print("That's", gp, "grade points.")
```

Solution to Exercise 53: Assessing Employees

```
##
# Report whether an employee's performance is unacceptable, acceptable
# or meritorious based on the rating entered by the user.
#
RAISE_FACTOR = 2400.00
UNACCEPTABLE = 0
ACCEPTABLE = 0.4
MERITORIOUS = 0.6

# Read the rating from the user
rating = float(input("Enter the rating: "))

# Classify the performance
if rating == UNACCEPTABLE:
  performance = "Unacceptable"
elif rating == ACCEPTABLE:
  performance = "Acceptable"
elif rating >= MERITORIOUS:
  performance = "Meritorious"
else:
  performance = ""

# Report the result
if performance == "":
  print("That wasn't a valid rating.")
else:
  print("Based on that rating, the performance is %s." % performance)
  print("You will receive a raise of $%.2f." % (rating * RAISE_FACTOR))
```

The parentheses around `rating * RAISE_FACTOR` on the final line are necessary because the `%` and `*` operators have the same precedence. Including the parentheses forces Python to perform the mathematical calculation before formatting the result.

Solution to Exercise 57: Is it a Leap Year?

```
##
#  Determine whether or not a year is a leap year.
#

# Read the year from the user
year = int(input("Enter a year: "))

# Determine if it is a leap year
if year % 400 == 0:
  isLeapYear = True
elif year % 100 == 0:
  isLeapYear = False
elif year % 4 == 0:
  isLeapYear = True
else:
  isLeapYear = False

# Display the result
if isLeapYear:
  print(year, "is a leap year.")
else:
  print(year, "is not a leap year.")
```

Solution to Exercise 59: Is a License Plate Valid?

```
## Determine whether or not a license plate is valid.  A valid license plate either consists of:
#    1)  3 letters followed by 3 numbers, or
#    2)  4 numbers followed by 3 numbers

# Read the plate from the user
plate = input("Enter the license plate: ")

# Check the status of the plate and display it.  It is necessary to check each of the 6 characters
# for an older style plate, or each of the 7 characters for a newer style plate.
if len(plate) == 6 and plate[0] >= "A" and plate[0] <= "Z" and \
  plate[1] >= "A" and plate[1] <= "Z" and \
  plate[2] >= "A" and plate[2] <= "Z" and \
  plate[3] >= "0" and plate[3] <= "9" and \
  plate[4] >= "0" and plate[4] <= "9" and \
  plate[5] >= "0" and plate[5] <= "9":
    print("The plate is a valid older style plate.")
elif len(plate) == 7 and plate[0] >= "0" and plate[0] <= "9" and \
  plate[1] >= "0" and plate[1] <= "9" and \
  plate[2] >= "0" and plate[2] <= "9" and \
  plate[3] >= "0" and plate[3] <= "9" and \
  plate[4] >= "A" and plate[4] <= "Z" and \
  plate[5] >= "A" and plate[5] <= "Z" and \
  plate[6] >= "A" and plate[6] <= "Z":
    print("The plate is a valid newer style plate.")
```

```
else:
  print("The plate is not valid.")
```

Solution to Exercise 60: Roulette Payouts

```
##
# Display the bets that pay out in a roulette simulation.
#
from random import randrange

# Simulate spinning the wheel, using 37 to represent 00
value = randrange(0, 38)
if value == 37:
  print("The spin resulted in 00...")
else:
  print("The spin resulted in %d..." % value)

# Display the payout for a single number
if value == 37:
  print("Pay 00")
else:
  print("Pay", value)

# Display the color payout
# The first line in the condition checks for 1, 3, 5, 7, 9
# The second line in the condition checks for 12, 14, 16, 18
# The third line in the condition checks for 19, 21, 23, 25, 27
# The fourth line in the condition checks for 30, 32, 34, 36
if value % 2 == 1 and value >= 1 and value <= 9 or \
   value % 2 == 0 and value >= 12 and value <= 18 or \
   value % 2 == 1 and value >= 19 and value <= 27 or \
   value % 2 == 0 and value >= 30 and value <= 36:
  print("Pay Red")
elif value == 0 or value == 37:
  pass
else:
  print("Pay Black")

# Display the odd vs. even payout
if value >= 1 and value <= 36:
  if value % 2 == 1:
    print("Pay Odd")
  else:
    print("Pay Even")

# Display the lower number vs. upper number payout
if value >= 1 and value <= 18:
  print("Pay 1 to 18")
elif value >= 19 and value <= 36:
  print("Pay 19 to 36")
```

The body of an `if`, `elif` or `else` must contain at least one statement. The `pass` statement can be used in situations where a statement is required but there is no work to be performed.

Loop Solutions

11

Solution to Exercise 64: No more Pennies

```
##
#  Compute the total due when several items are purchased.  The amount
#  payable for cash transactions is rounded to the closest nickel because
#  pennies have been phased out in Canada.
#
PENNIES_PER_NICKEL = 5
NICKEL = 0.05
```

> While it is highly unlikely that the number of pennies in a nickel will ever change, it is possible (even likely) that we will need to update our program at some point in the future so that it rounds to the closest dime. Using constants will make it easier to perform that update when it is needed.

```
# Track the total of all the items
total = 0.00

# Read the price of the first item as a string
line = input("Enter the price of the item (blank to quit): ")

# Continue reading items until a blank line is entered
while line != "":
  # Add the cost of the item to the total (after converting it to a float)
  total = total + float(line)

  # Read the cost of the next item
  line = input("Enter the price of the item (blank to quit): ")

# Display the exact total payable
print("The exact amount payable is %.02f" % total)

# Compute the number of pennies that would be left if the total was paid
# only using nickels
rounding_indicator = total * 100 % PENNIES_PER_NICKEL
```


B. Stephenson, *The Python Workbook*, DOI 10.1007/978-3-319-14240-1_11

```
if rounding_indicator < PENNIES_PER_NICKEL / 2:
  # If the number of pennies left is less than 2.5 then we round down by
  # subtracting that number of pennies from the total
  cash_total = total - rounding_indicator / 100
else:
  # Otherwise we add a nickel and then subtract the number of pennies
  cash_total = total + NICKEL - rounding_indicator / 100

# Display the cash amount payable
print("The cash amount payable is %.02f" % cash_total)
```

Solution to Exercise 65: Computer the Perimeter of a Polygon

```
##
# Compute the perimeter of a polygon. The user will enter a blank line
# for the x-coordinate to indicate that all of the points have been entered.
#
from math import sqrt

# Store the perimeter of the polygon
perimeter = 0

# Read the coordinate of the first point
first_x = float(input("Enter the x part of the coordiante: "))
first_y = float(input("Enter the y part of the coordinate: "))

# Provide initial values for prev_x and prev_y
prev_x = first_x
prev_y = first_y

# Read the remaining coordinates
line = input("Enter the x part of the coordinate (blank to quit): ")
while line != "":
  # Convert the x part to a number and read the y part
  x = float(line)
  y = float(input("Enter the y part of the coordinate: "))

  # Compute the distance to the previous point and add it to the perimeter
  dist = sqrt((prev_x - x) ** 2 + (prev_y - y) ** 2)
  perimeter = perimeter + dist

  # Set up prev_x and prev_y for the next loop iteration
  prev_x = x
  prev_y = y

  # Read the x part of the next coordinate
  line = input("Enter the x part of the coordinate (blank to quit): ")

# Compute the distance from the last point to the first point and add it to the perimeter
dist = sqrt((first_x - x) ** 2 + (first_y - y) ** 2)
perimeter = perimeter + dist
```

The distance between the points is computed using Pythagorean theorem.

```
# Display the result
print("The perimeter of that polygon is", perimeter)
```

Solution to Exercise 67: Admission Price

```
##
#  Compute the admission price for a group visiting the zoo.
#

# Store the admission prices as constants
BABY_PRICE = 0.00
CHILD_PRICE = 14.00
ADULT_PRICE = 23.00
SENIOR_PRICE = 18.00

# Store the age limits as constants
BABY_LIMIT = 2
CHILD_LIMIT = 12
ADULT_LIMIT = 64

# Create a variable to hold the total admission cost for all guests
total = 0

# Keep on reading ages until the user enters a blank line
line = input("Enter the age of the guest (blank to finish): ")
while line != "":
  age = int(line)

  # Add the correct amount to the total
  if age <= BABY_LIMIT:
    total = total + BABY_PRICE
  elif age <= CHILD_LIMIT:
    total = total + CHILD_PRICE
  elif age <= ADULT_LIMIT:
    total = total + ADULT_PRICE
  else:
    total = total + SENIOR_PRICE

  # Read the next line from the user
  line = input("Enter the age of the guest (blank to finish): ")

# Display the total due for the group, formatted using two decimal places
print("The total for that group is $%.2f" % total)
```

The first condition is not necessary with the current admission prices. However, including it makes it easy to start charging for babbies in the future.

Solution to Exercise 68: Parity Bits

```
##
#  Compute even parity for sets of 8 bits entered by the user.
#
```

```
# Read the first line of input
line = input("Enter 8 bits: ")

# Continue looping until a blank line is entered
while line != "":
  # Ensure that the line has a total of 8 zeros and ones and exactly 8 characters
  if line.count("0") + line.count("1") != 8 or len(line) != 8:
    # Display an appropriate error message
    print("That wasn't 8 bits... Try again.")
  else:
    # Count the number of ones
    ones = line.count("1")

    # Display the parity bit
    if ones % 2 == 0:
      print("The parity bit should be 0.")
    else:
      print("The parity bit should be 1.")

  # Read the next line of input
  line = input("Enter 8 bits: ")
```

The `count` method returns the number of times that the string provided as a parameter occurs in the string on which the method is invoked.

Solution to Exercise 70: Caesar Cipher

```
##
#  Implement a Caesar cipher that shifts all of the letters in a message by an amount
#  provided by the user.  Use a negative shift value to decode a message.
#

# Read the message and shift amount from the user
message = input("Enter the message: ")
shift = int(input("Enter the shift value: "))

# Process each character, constructing a new message
new_message = ""
for ch in message:
  if ch >= "a" and ch <= "z":
    # Process a lowercase letter by determining its
    # position in the alphabet (0 - 25), computing its
    # new position, and adding it to the new message
    pos = ord(ch) - ord("a")
    pos = (pos + shift) % 26
    new_char = chr(pos + ord("a"))
    new_message = new_message + new_char
  elif ch >= "A" and ch <= "Z":
    # Process an uppercase letter by determining its position in the alphabet
    # (0 - 25), computing its new position, and adding it to the new message
    pos = ord(ch) - ord("A")
    pos = (pos + shift) % 26
    new_char = chr(pos + ord("A"))
    new_message = new_message + new_char
```

The `ord` function converts a character to its integer position within the ASCII table. The `chr` function returns the character from the ASCII table in the position provided.

```
  else:
    # If the character is not a letter then copy it into the new message
    new_message = new_message + ch

# Display the shifted message
print("The shifted message is", new_message)
```

Solution to Exercise 72: Is a String a Palindrome?

```
##
#  Determine whether or not a string entered by the user is a palindrome.
#

# Read the input from the user
line = input("Enter a string: ")

# Assume that it is a palindrome until we can prove otherwise
is_palindrome = True

# Check the characters, starting from the ends until
# the middle is reached
for i in range(0, len(line) // 2):
  # If the characters don't match then mark
  # that the string is not a palindrome
  if line[i] != line[len(line) - i - 1]:
    is_palindrome = False

# Display a meaningful output message
if is_palindrome:
  print(line, "is a palindrome")
else:
  print(line, "is not a palindrome")
```

All of the parameters to the range function must be integers. The // operator is used so that the result of the division is truncated to an integer.

Solution to Exercise 74: Multiplication Table

```
##
# Display a multiplication table for 1 times 1 through 10 times 10.
#
MIN = 1
MAX = 10

# Display the top row of labels
print("    ", end="")
for i in range(MIN, MAX + 1):
  print("%4d" % i, end="")
print()
```

Including `end=""` as the final parameter to print prevents it from moving down to the next line after displaying the value.

```
# Display the table
for i in range(MIN, MAX + 1):
  print("%4d" % i, end="")
  for j in range(MIN, MAX + 1):
    print("%4d" % (i * j), end="")
  print()
```

Solution to Exercise 75: Greatest Common Divisor

```
##
#  Compute the greatest common divisor of two positive integers using a while loop.
#

# Read two positive integers from the user
n = int(input("Enter a positive integer: "))
m = int(input("Enter a positive integer: "))

# Initialize d to the smaller of n and m
d = min(n, m)

# Use a while loop to find the greatest common divisor of n and m
while n % d != 0 or m % d != 0:
  d = d - 1

# Report the result
print("The greatest common divisor of", n, "and", m, "is", d)
```

Solution to Exercise 78: Decimal to Binary

```
##
#  Convert a number from Decimal (base 10) to Binary (base 2)
#
NEW_BASE = 2

# Read the number to convert from the user
num = int(input("Enter a non-negative integer: "))

# Generate the binary representation of num,
# storing it in result
result = ""
q = num

# Perform the body of the loop once
r = q % NEW_BASE
result = str(r) + result
q = q // NEW_BASE
```

The algorithm provided for this question is expressed using a repeat-until loop. However, this structure isn't available in Python. As a result, the algorithm has to be adjusted so that it generates the same result using a `while` loop. This is achieved by duplicating the loop body and placing it ahead of the `while` loop.

```
# Keep on looping until q == 0
while q > 0:
  r = q % NEW_BASE
  result = str(r) + result
  q = q // NEW_BASE

# Display the result
print(num, "in Decimal is", result, "in Binary.")
```

Solution to Exercise 79: Maximum Integer

```
##
#  Find the maximum of 100 random integers, counting the number of times the
#  maximum value is updated during the process
#
from random import randrange

NUM_ITEMS = 100

# Generate the first number and display it
mx_value = randrange(1, NUM_ITEMS + 1)
print(mx_value)

# Count of the number of updates
num_updates = 0

# For each of the remaining numbers
for i in range(1, NUM_ITEMS):
  # Generate a new random number
  current = randrange(1, NUM_ITEMS + 1)

  # If the generated number is the largest one we have seen so far
  if current > mx_value:
    # Update the maximum and count the update
    mx_value = current
    num_updates = num_updates + 1
    # Display the number, noting that an update occurred
    print(current, "<== Update")
  else:
    # Display the number
    print(current)

# Display the summary results
print("The maximum value found was", mx_value)
print("The maximum value was updated", num_updates, "times")
```

Function Solutions

12

Solution to Exercise 84: Median of Three Values

```
##
# Compute and display the median of three values entered by the user.  This
# program includes two implementations of the median function that
# demonstrate different techniques for computing the median of three values.
#

## Compute the median of three values using if statements
# @param a the first value
# @param b the second value
# @param c the third value
# @return the median of values a, b and c
#
def median(a, b, c):
  if a < b and b < c or a > b and b > c:
    return b
  if b < a and a < c or b > a and a > c:
    return a
  if c < a and b < c or c > a and b > c:
    return c

## Compute the median of three values using the min and max functions
# and a little bit of arithmetic
# @param a the first value
# @param b the second value
# @param c the third value
# @return the median of values a, b and c
#
def alternateMedian(a, b, c):
  return a + b + c - min(a, b, c) - max(a, b, c)
```

Each function that you write should begin with a comment. Lines beginning with `@param` are used to describe the function's parameters. The value returned by the function is describe by a line that begins with `@return`.

The median of three values is the sum of the values, less the smallest, less the largest.

B. Stephenson, *The Python Workbook*, DOI 10.1007/978-3-319-14240-1_12

```
# Display the median of 3 values entered by the user
def main():
  x = float(input("Enter the first value: "))
  y = float(input("Enter the second value: "))
  z = float(input("Enter the third value: "))

  print("The median value is:", median(x, y, z))
  print("Using the alternative method, it is:", alternateMedian(x, y, z))

# Call the main function
main()
```

Solution to Exercise 86: The Twelve days of Christmas

```
##
# Generate the complete lyrics for the song The Twelve Days of Christmas.
#
from int_ordinal import intToOrdinal
```

The function that was written for the previous exercise is imported into this program so that the code for converting from an integer to its ordinal number does not have to be duplicated here.

```
## Generate and display one verse of The Twelve Days of Christmas
# @param n the verse to generate
# @return (none)
def displayVerse(n):
  print("One the", intToOrdinal(n), "day of Christmas")
  print("my true love sent to me:")

  if n >= 12:
    print("Twelve drummers drumming,")
  if n >= 11:
    print("Eleven pipers piping,")
  if n >= 10:
    print("Ten lords a leaping,")
  if n >= 9:
    print("Nine ladies dancing,")
  if n >= 8:
    print("Eight maids a milking,")
  if n >= 7:
    print("Seven swans a swimming,")
  if n >= 6:
    print("Six geese a laying,")
  if n >= 5:
    print("Five golden rings,")
  if n >= 4:
    print("Four calling birds,")
  if n >= 3:
    print("Three French hens,")
```

```
  if n >= 2:
      print("Two turtle doves,")
  if n == 1:
    print("A", end=" ")
  else:
    print("And a", end=" ")
  print("partridge in a pear tree.")
  print()

# Display all 12 verses of the song
def main():
  for verse in range(1, 13):
    displayVerse(verse)

# Call the main function
main()
```

Solution to Exercise 87: Center a String in the Terminal

```
##
# Center a string of characters within a certain width.
#
WIDTH = 80

## Create a new string that will be centered within a given width when it is printed.
# @param s the string that will be centered
# @param width the width in which the string will be centered
# @return a new copy of s that contains the leading spaces needed so that
#       s will appear centered when it is printed.
def center(s, width):
  # If the string is too long to center, then the original string is returned
  if width < len(s):
    return s

  # Compute the number of spaces needed and generate the result
  spaces = (width - len(s)) // 2
  result = " " * spaces + s

  return result

# Demonstrate the center function
def main():
  print(center("A Famous Story", WIDTH))
  print(center("by:", WIDTH))
  print(center("Someone Famous", WIDTH))
  print()
  print("Once upon a time...")

# Call the main function
main()
```

The // operator is used so that the result of the division will be truncated to an integer. This is necessary because a string can only be replicated an integer number of times.

Solution to Exercise 89: Capitalize it

```
##
# Improve the capitalization of a string by replacing " i " with " I " and by
# capitalizing the first letter of each sentence.
#

## Capitalize the appropriate characters in a string
#  @param s the string that needs capitalization
#  @return a new string with the capitalization improved
def capitalize(s):
  # Correct the capitalization for i
  result = s.replace(" i ", " I ")

  # Capitalize the first character of the string
  if len(s) > 0:
    result = result[0].upper() + \
             result[1 : len(result)]

  # Capitalize the first letter that follows a ".", "!" or "?"
  pos = 0
  while pos < len(s):
    if result[pos] == "." or result[pos] == "!" or result[pos] == "?":
      # Move past the ".", "!" or "?"
      pos = pos + 1

      # Move past any spaces
      while pos < len(s) and result[pos] == " ":
        pos = pos + 1

      # If we haven't reached the end of the string then replace
      # the current character with its uppercase equivalent
      if pos < len(s):
        result = result[0 : pos] + \
                 result[pos].upper() + \
                 result[pos + 1 : len(result)]

    # Move to the next character
    pos = pos + 1

  return result

# Demonstrate the capitalize function
def main():
  s = input("Enter some text: ")
  capitalized = capitalize(s)
  print("It is capitalized as:", capitalized)

# Call the main function
main()
```

The `replace` method replaces all occurences of its first parameter with its second parameter in the string on which it is invoked.

Using a colon inside of square brackets retrieves a portion of a string. The characters that are retrieved start at the position that appears to the left of the colon, going up to (but not including) the position that appears to the right of the colon.

Solution to Exercise 90: Does a String Represent an Integer?

```
##
# Determine whether or not a string entered by the user is an integer.
#

## Determine if a string contains a valid representation of an integer
# @param s the string to check
# @return True if s represents an integer. False otherwise.
#
def isInteger(s):
  # Remove whitespace from the beginning and end of the string
  s = s.strip()

  # Determine if the remaining characters form a valid integer
  if (s[0] == "+" or s[0] == "-") and s[1:].isdigit():
    return True
  if s.isdigit():
    return True
  return False

# Demonstrate the isInteger function
def main():
  s = input("Enter a string: ")
  if isInteger(s):
    print("That string represents an integer.")
  else:
    print("That string does not represent an integer.")

# Only call the main function when this file has not been imported
if __name__ == "__main__":
  main()
```

The `isdigit` method returns true if and only if the string is at least one character in length and all of the characters in the string are digits.

The `__name__` variable is automatically assigned a value by Python when the program starts running. It contains `"__main__"` when the file is executed directly by Python. It contains the name of the module when the file is imported into another program.

Solution to Exercise 92: Is a Number Prime?

```
##
# Determine if a number entered by the user is prime.
#

## Determine whether or not a number is prime
# @param n the integer to test
# @return True if the number is prime, False otherwise
def isPrime(n):
  if n <= 1:
    return False
```

```
  # Check each number from 2 up to but not including n to see if it divides evenly into n
  for i in range(2, n):
    if n % i == 0:
      return False
  return True

# Determine if a number entered by the user is prime
def main():
  value = int(input("Enter an integer: "))
  if isPrime(value):
    print(value,"is prime.")
  else:
    print(value,"is not prime.")

# Call the main function if the file has not been imported
if __name__ == "__main__":
  main()
```

If `n % i == 0` then `n` is evenly divisible by `i`, indicating that `n` is not prime.

Solution to Exercise 94: Random Password

```
##
# Generate and display a random password containing between 7 and 10 characters.
#
from random import randint

SHORTEST = 7
LONGEST = 10
MIN_ASCII = 33
MAX_ASCII = 126

## Generate a random password
# @return a string containing a random password
def randomPassword():
  # Select a random length for the password
  randomLength = randint(SHORTEST, LONGEST)

  # Generate an appropriate number of random characters, adding each one to the end of result
  result = ""
  for i in range(randomLength):
    randomChar = chr(randint(MIN_ASCII, MAX_ASCII))
    result = result + randomChar

  # Return the random password
  return result

# Generate and display a random password
def main():
  print("Your random password is:", randomPassword())

# Call the main function only if the module is not imported
if __name__ == "__main__":
  main()
```

The `chr` function takes an ASCII code as its parameter. It returns a string containing the character with that ASCII code as its result.

Solution to Exercise 96: Check a Password

```
##
# Check whether or not a password is good.
#

## Check whether or not a password is good. A good password is at least 8 characters
# long and contains an uppercase letter, a lowercase letter and a number.
# @param password the password to check
# @return True if the password is good, False otherwise
def checkPassword(password):
  has_upper = False
  has_lower = False
  has_num = False

  # Check each character in the password and see which requirement it meets
  for ch in password:
    if ch >= "A" and ch <= "Z":
      has_upper = True
    elif ch >= "a" and ch <= "z":
      has_lower = True
    elif ch >= "0" and ch <= "9":
      has_num = True

  # If the password has all 4 properties
  if len(password) >= 8 and has_upper and has_lower and has_num:
    return True

  # The password is missing at least on property
  return False

# Demonstrate the password checking function
def main():
  p = input("Enter a password: ")
  if checkPassword(p):
    print("That's a good password.")
  else:
    print("That isn't a good password.")

# Call the main function only if the file has not been imported into another program
if __name__ == "__main__":
  main()
```

Solution to Exercise 99: Arbitrary Base Conversions

```
##
# Convert a number from one base to another. Both the source base and the
# destination base must be between 2 and 16.
#
from hex_digit import *
```

The `hex_digit` module contains the functions `hex2int` and `int2hex` which were developed while solving Exercise 98. Using `import *` imports all of the functions from that module.

```
## Convert a number from base 10 to base n
#  @param num the base 10 number to convert
#  @param new_base the base to convert to
#  @return the string of digits in new_base
def dec2n(num, new_base):
  # Generate the representation of num in base new_base, storing it in result
  result = ""
  q = num

  # Perform the body of the loop once
  r = q % new_base
  result = int2hex(r) + result
  q = q // new_base

  # Continue looping until q == 0
  while q > 0:
    r = q % new_base
    result = int2hex(r) + result
    q = q // new_base

  # Return the result
  return result
```

```
## Convert a number from base b to base 10
#  @param num the base b number, stored in a string
#  @param b the base of the number to convert
#  @return the base 10 number
def n2dec(num, b):
  decimal = 0
  power = 0

  # Process each digit in the base b number
  for i in range(len(num)):
    decimal = decimal * b
    decimal = decimal + hex2int(num[i])

  # Return the result
  return decimal
```

The base *b* number must be stored in a string because it may contain letters that represent digits in bases larger than 10.

```
# Convert a number between two arbitrary bases
def main():
  # Read the number from the user
  from_base = int(input("Enter the base to convert from: "))
  from_num = input("Enter a sequence of digits in that base: ")
```

```
# Convert to base 10 and display the result
dec = n2dec(from_num, from_base)
print("That's %d in base 10." % dec)

# Convert to a new base and display the result
to_base = int(input("Enter the base to convert to: "))
to_num = dec2n(dec, to_base)
print("That's %s in base %d." % (to_num, to_base))
```

Solution to Exercise 101: Reduce a Fraction to Lowest Terms

```
##
# Reduce a fraction to its lowest terms.
#

## Compute the greatest common divisor of two integers.
# @param n the first integer under consideration (must be non-zero)
# @param m the second integer under consideration (must be non-zero)
# @return the greatest common divisor of the integers
def gcd(n, m):
  # Initialize d to the smaller of n and m
  d = min(n, m)

  # Use a while loop to find the greatest common divisor of n and m
  while n % d != 0 or m % d != 0:
    d = d - 1

  return d
```

This function used a loop to achieve its goal. There is also an elegant solution for finding the greatest common divisor of two integers that uses recursion. The recursive solution is explored in a later exercise.

```
## Reduce a fraction to lowest terms.
# @param the integer numerator of the fraction
# @param the integer denominator of the fraction (must be non-zero)
# @return the numerator and denominator of the reduced fraction
def reduce(num, den):
  # If the numerator is 0 then the reduced fraction is 0 / 1
  if num == 0:
    return (0, 1)

  # Compute the greatest common divisor of the numerator and denominator
  g = gcd(num, den)
```

```
  # Divide both the numerator and denominator
  # by the gcd and return the result
  return (num // g, den // g)

# Read the fraction from the user and display the reduced fraction
def main():
  # Read the input from the user
  num = int(input("Enter the numerator: "))
  den = int(input("Enter the denominator: "))

  # Compute the reduced fraction
  (n, d) = reduce(num, den)

  # Display the result
  print("%d/%d can be reduced to %d/%d." % (num, den, n, d))

# Call the main function
main()
```

We have used the integer division operator, `//`, so that we return a result like `(1, 3)` instead of `(1.0, 3.0)`.

Solution to Exercise 102: Reduce Measures

```
##
# Reduce an imperial measurement so that it is expressed using the largest possible units of
# measure. For example, 59 teaspoons is reduced to 1 cup, 3 tablespoons, 2 teaspoons.
#
TSP_PER_TBSP = 3
TSP_PER_CUP = 48

## Reduce an imperial measurement so that it is expressed using the largest possible
# units of measure.
# @param num the number of units that need to be reduced
# @param unit the unit of measure (cup, tablespoon or teaspoon)
# @return a string representing the measurement in reduced form
def reduceMeasure(num, unit):
  # Compute the number of teaspoons that the parameters represent
  unit = unit.lower()
```

The unit is converted to lowercase by invoking the `lower` method on `unit` and storing the result into the same variable. This allows the user to use any mixture of uppercase and lowercase letters when specifying the unit.

```
  if unit == "teaspoon" or unit == "teaspoons":
    teaspoons = num
  elif unit == "tablespoon" or unit == "tablespoons":
    teaspoons = num * TSP_PER_TBSP
  elif unit == "cup" or unit == "cups":
    teaspoons = num * TSP_PER_CUP
```

```
# Convert the number of teaspoons to the largest possible units of measure
cups = teaspoons // TSP_PER_CUP
teaspoons = teaspoons - cups * TSP_PER_CUP
tablespoons = teaspoons // TSP_PER_TBSP
teaspoons = teaspoons - tablespoons * TSP_PER_TBSP

# Generate the result string
result = ""

# Add the number of cups to the result string (if any)
if cups > 0:
  result = result + str(cups) + " cup"
  # Make cup plural if there is more than one
  if cups > 1:
    result = result + "s"

# Add the number of tablespoons to the result string (if any)
if tablespoons > 0:
  # Include a comma if there were some cups
  if result != "":
    result = result + ", "

  result = result + str(tablespoons) + " tablespoon"
  # Make tablespoon plural if there is more than one
  if tablespoons > 1:
    result = result + "s"

# Add the number of teaspoons to the result string (if any)
if teaspoons > 0:
  # Include a comma if there were some cups and/or tablespoons
  if result != "":
    result = result + ", "

  result = result + str(teaspoons) + " teaspoon"
  # Make teaspoons plural if there is more than one
  if teaspoons > 1:
    result = result + "s"

# Handle the case where the number of units was 0
if result == "":
  result = "0 teaspoons"


return result
```

Several test cases are included in this program. They exercise a variety of different combinations of zero, one and multiple occurences of the different units of measure. While these test cases are reasonably thorough, they do not guarantee that the program is bug free.

```
# Demonstrate the reduceMeasure function by performing several reductions
def main():
  print("59 teaspoons is %s." % reduceMeasure(59, "teaspoons"))
  print("59 tablespoons is %s." % reduceMeasure(59, "tablespoons"))
  print("1 teaspoon is %s." % reduceMeasure(1, "teaspoon"))
  print("1 tablespoon is %s." % reduceMeasure(1, "tablespoon"))
  print("1 cup is %s." % reduceMeasure(1, "cup"))
  print("4 cups is %s." % reduceMeasure(4, "cups"))
  print("3 teaspoons is %s." % reduceMeasure(3, "teaspoons"))
  print("6 teaspoons is %s." % reduceMeasure(6, "teaspoons"))
  print("95 teaspoons is %s." % reduceMeasure(95, "teaspoons"))
  print("96 teaspoons is %s." % reduceMeasure(96, "teaspoons"))
  print("97 teaspoons is %s." % reduceMeasure(97, "teaspoons"))
  print("98 teaspoons is %s." % reduceMeasure(98, "teaspoons"))
  print("99 teaspoons is %s." % reduceMeasure(99, "teaspoons"))

# Call the main function
main()
```

Solution to Exercise 103: Magic Dates

```
##
# Determine all of the magic dates in the 1900s
#
from days_in_month import daysInMonth

## Determine whether or not a date is ''magic''
# @param day the day portion of the date
# @param month the month portion of the date
# @param year the year portion of the date
# @return True if the date is magic, False otherwise
def isMagicDate(day, month, year):
  if day * month == year % 100:
    return True

  return False

# Find and display all of the magic dates in the 1900s
def main():
  for year in range(1900, 2000):
    for month in range(1, 13):
      for day in range(1, daysInMonth(month, year) + 1):
        if isMagicDate(day, month, year):
          print("%02d/%02d/%04d is a magic date." % (day, month, year))

# Call the main function
main()
```

The expression `year % 100` evaluates to the two digit year.

List Solutions

13

Solution to Exercise 104: Sorted Order

```
##
#  Display the integers entered by the user in ascending order.
#

# Start with an empty list
data = []

# Read values, adding them to the list, until the user enters 0
num = int(input("Enter an integer (0 to quit): "))
while num != 0:
  data.append(num)
  num = int(input("Enter an integer (0 to quit): "))

# Sort the values
data.sort()
```

Invoking the `sort` method on a list rearranges the elements in the list into sorted order. Using the `sort` method is appropriate for this problem because there is no need to keep a copy of the original list. The `sorted` function can be used to create a new copy of the list where the elements are in sorted order. Calling the `sorted` function does not modify the original list. As a result, it can be used in situations where the original list and the sorted list are needed simultaneously.

```
# Display the values in ascending order
print("The values, sorted into ascending order, are:")
for num in data:
  print(num)
```


B. Stephenson, *The Python Workbook*, DOI 10.1007/978-3-319-14240-1_13

Solution to Exercise 106: Remove Outliers

```
##
# Remove the outliers from a data set.
#

## Remove the outliers from a list of data
# @param data the list of data values to process
# @param num_outliers the number of smallest and largest values to remove
# @return a new copy of data where the values are sorted into ascending
#       order and the smallest and largest values have been removed
def removeOutliers(data, num_outliers):
  # Create a new copy of the list that is in sorted order
  retval = sorted(data)

  # Remove num_outliers largest values
  for i in range(num_outliers):
    retval.pop()

  # Remove num_outliers smallest values
  for i in range(num_outliers):
    retval.pop(0)

  # Return the result
  return retval

# Read data from the user, and remove the two largest and two smallest values
def main():
  # Read values from the user until a blank line is entered
  values = []
  s = input("Enter a value (blank line to quit): ")
  while s != "":
    num = float(s)
    values.append(num)
    s = input("Enter a value (blank line to quit): ")

  # Display the result or an appropriate error message
  if len(values) < 4:
    print("You didn't enter enough values.")
  else:
    print("With the outliers removed: ", removeOutliers(values, 2))
    print("The original data: ", values)

# Call the main function
main()
```

The smallest and largest outliers could be removed using the same loop. Two loops are used in this solution to make the steps more clear.

Solution to Exercise 107: Avoiding Duplicates

```
##
#  Read a collection of words entered by the user.  Display each word entered
#  by the user only once, in the same order that the words were entered.
#

# Begin reading words into a list
words = []
word = input("Enter a word (blank line to quit): ")
while word != "":
  # Only add the word to the list if
  # it is not already present in it
  if word not in words:
    words.append(word)

  # Read the next word from the user
  word = input("Enter a word (blank line to quit): ")

# Display the unique words
for word in words:
  print(word)
```

The expression `word not in words` is equivalent to `not (word in words)`.

Solution to Exercise 108: Negatives, Zeros and Positives

```
##
#  Read a collection of integers from the user.  Display all of the negative numbers,
#  followed by all of the zeros, followed by all of the positive numbers.
#

# Create three lists to store the negative, zero and
# positive values
negatives = []
zeros = []
positives = []

# Read all of the integers from the user, storing each
# integer in the correct list
line = input("Enter an integer (blank to quit): ")
while line != "":
  num = int(line)

  if num < 0:
    negatives.append(num)
  elif num > 0:
    positives.append(num)
  else:
    zeros.append(num)

  # Read the next line of input from the user
  line = input("Enter an integer (blank to quit): ")
```

This solution uses a list to keep track of the zeros that are entered. However, because all of the zeros are the same, it isn't actually necessary to save them. Instead, one could use an integer variable to count the number of zeros and then display that many zeros later in the program.

```
# Display all of the negative values, then all of the zeros, then all of the positive values
print("The numbers were: ")

for n in negatives:
  print(n)

for n in zeros:
  print(n)

for n in positives:
  print(n)
```

Solution to Exercise 110: Perfect Numbers

```
##
# A number, n, is a perfect number if the sum of the proper divisors of n is equal
# to n. This program displays all of the perfect numbers between 1 and LIMIT.
#
from proper_divisors import properDivisors

LIMIT = 10000

## Determine whether or not a number is perfect. A number is perfect if the
# sum of its proper divisors is equal to the number itself.
# @param n the number to check for perfection
# @return True if the number is perfect, False otherwise
def isPerfect(n):
  # Get a list of the proper divisors of n
  divisors = properDivisors(n)

  # Compute the total of all of the divisors
  total = 0
  for d in divisors:
    total = total + d

  # Return the appropriate result
  if total == n:
    return True
  return False

# Display all of the perfect numbers between 1 and LIMIT
def main():
  print("The perfect numbers between 1 and", LIMIT, "are:")
  for i in range(1, LIMIT + 1):
    if isPerfect(i):
      print(" ", i)

# Call the main function
main()
```

The total could also be computed using the `sum` function. This would reduce the calculation of the total to a single line.

Solution to Exercise 113: Formatting a List

```
##
#  Display a list of items so that they are separated by commas and the word
#  ''and'' appears between the final two items.
#

## Format a list of items so that they are separated by commas and ''and''
# @param items the list of items to format
# @return a string containing the items with the desired formatting
#
def formatList(items):
  # Handle lists of 0 and 1 items as special cases
  if len(items) == 0:
    return "<empty>"
  if len(items) == 1:
    return str(items[0])

  # Loop over all of the items in the list except the last two
  result = ""
  for i in range(0, len(items) - 2):
    result = result + str(items[i]) + ", "
```

Each item is explicitly cast to a string by calling the `str` function before it is added to the result. This allows `formatList` to format lists that contain numbers in addition to strings.

```
  # Add the second last and last items to the result, separated by ''and''
  result = result + str(items[len(items) - 2]) + " and "
  result = result + str(items[len(items) - 1])

  # Return the result
  return result

##
#  Read several items entered by the user and display them with nice formatting.
#
def main():
  # Read items from the user until a blank line is entered
  items = []
  line = input("Enter an item (blank to quit): ")
  while line != "":
    items.append(line)
    line = input("Enter an item (blank to quit): ")

  # Format and display the items
  print("The items are %s." % formatList(items))

# Call the main function
main()
```

Solution to Exercise 114: Random Lottery Numbers

```
##
# Compute random but distinct numbers for a lottery ticket.
#
from random import randrange

MIN_NUM = 1
MAX_NUM = 49
NUM_NUMS = 6

# Keep a list of the numbers for the ticket
ticket_nums = []

# Generate NUM_NUMS random but distinct numbers
for i in range(NUM_NUMS):
  # Generate a number that isn't already on the ticket
  rand = randrange(MIN_NUM, MAX_NUM + 1)
  while rand in ticket_nums:
    rand = randrange(MIN_NUM, MAX_NUM + 1)

  # Add the distinct number to the ticket
  ticket_nums.append(rand)

# Sort the numbers into ascending order and display them
ticket_nums.sort()
print("Your numbers are: ", end="")
for n in ticket_nums:
  print(n, end=" ")
print()
```

Using constants makes it easy to recongifure our program for other lotteries.

Solution to Exercise 118: Shuffling a Deck of Cards

```
##
# Create a deck of cards and shuffle it.
#
from random import randrange

# Construct a standard deck of cards with 4 suits and 13 values per suit
# @return a list of cards, with each card represented by two characters
def createDeck():
  # Create a list to store the cards in
  cards = []

  # For each suit and each value
  for suit in ["s", "h", "d", "c"]:
    for value in ["2", "3", "4", "5", "6", "7", "8", "9", \
                  "T", "J", "Q", "K", "A"]:
      # Construct the card and add it to the list
      cards.append(value + suit)
```

```
  # Return the complete deck of cards
  return cards

# Shuffle a deck of cards, modifying the deck of cards passed as a parameter
# @param cards the list of cards to shuffle
def shuffle(cards):
  # For each card
  for i in range(0, len(cards)):
    # Pick a random index
    other_pos = randrange(0, len(cards))

    # Swap the current card with the one at the random position
    temp = cards[i]
    cards[i] = cards[other_pos]
    cards[other_pos] = temp

# Display a deck of cards before and after it has been shuffled
def main():
  cards = createDeck()
  print("The original deck of cards is: ")
  print(cards)
  print()

  shuffle(cards)
  print("The shuffled deck of cards is: ")
  print(cards)

# Call the main program only if this file has not been imported
if __name__ == "__main__":
  main()
```

Solution to Exercise 121: Count the Elements

```
##
#  Count the number of elements in a list that are greater than or equal
#  to some minimum value and less than some maximum value.
#

# Determine how many elements in data are greater than or equal to mn and less than mx.
# @param data the list to process
# @param mn the minimum acceptable value
# @param mx the exclusive upper bound on acceptability
# @return the number of elements, e, such that mn <= e < mx
def countRange(data, mn, mx):
  # Count the number of elements within the acceptable range
  count = 0
  for e in data:
    # Check each element
    if mn <= e and e < mx:
      count = count + 1

  # Return the result
  return count
```

```
# Demonstrate the countRange function
def main():
  data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

  # Test a case where some elements are within the range
  print("Counting the elements in [1..10] that are between 5 and 7...")
  print("Result: %d  Expected: 2" % countRange(data, 5, 7))

  # Test a case where all elements are within the range
  print("Counting the elements in [1..10] that are between -5 and 77...")
  print("Result: %d  Expected: 10" % countRange(data, -5, 77))

  # Test a case where no elements are within the range
  print("Counting the elements in [1..10] that are between 12 and 17...")
  print("Result: %d  Expected: 0" % countRange(data, 12, 17))

  # Test a case where the list is empty
  print("Counting the elements in [] that are between 0 and 100...")
  print("Result: %d  Expected: 0" % countRange([], 0, 100))

  # Test a case with duplicate values
  data = [1, 2, 3, 4, 1, 2, 3, 4]
  print("Counting the elements in [1, 2, 3, 4, 1, 2, 3, 4] that are", \
        "between 2 and 4...")
  print("Result: %d  Expected: 4" % countRange(data, 2, 4))

# Call the main program
main()
```

Solution to Exercise 122: Tokenizing a String

```
##
#  Tokenize a string containing a mathematical expression.
#

# Convert a mathematical expression into a list of tokens
# @param s the string to tokenize
# @return a list of the tokens in s, or an empty list if an error occurs
def tokenList(s) :
  # Remove all of the spaces from s
  s = s.replace(" ", "")
```

```
  # Loop through all of the characters in the string,
  # identifying the tokens and adding them to the list.
  tokens = []
  i = 0
  while i < len(s):
    # Handle the tokens that are always a single character: *, /, ^, ( and )
    if s[i] == "*" or s[i] == "/" or s[i] == "^" or \
       s[i] == "(" or s[i] == ")":
      tokens.append(s[i])
      i = i + 1

    # Handle + and -
    elif s[i] == "+" or s[i] == "-":
      # If there is a previous character and it is a number or close bracket
      # then the + or - is a token on its own
      if i > 0 and (s[i-1] >= "0" and s[i-1] <= "9" or s[i-1] == ")"):
        tokens.append(s[i])
        i = i + 1
      else:
        # The + or - is part of a number
        num = s[i]
        i = i + 1

        # Keep on adding characters to the token as long as they are digits
        while i < len(s) and s[i] >= "0" and s[i] <= "9":
          num = num + s[i]
          i = i + 1
        tokens.append(num)

    # Handle a number without a leading + or -
    elif s[i] >= "0" and s[i] <= "9":
      num = ""
      # Keep on adding characters to the token as long as they are digits
      while i < len(s) and s[i] >= "0" and s[i] <= "9":
        num = num + s[i]
        i = i + 1
      tokens.append(num)

    # Any other character means the expression is not valid.
    # Return an empty list to indicate that an error ocurred.
    else:
      return []
  return tokens

# Read an expression from the user and tokenize it, displaying the result
def main():
  exp = input("Enter a mathematical expression: ")
  tokens = tokenList(exp)
  print("The tokens are:", tokens)

# Call the main function only if the file hasn't been imported
if __name__ == "__main__":
  main()
```

Solution to Exercise 126: Generate All Sublists of a List

```
##
# Compute all sublists of a list
#

## Generate a list of of all of the sublists of a list
# @param data the list for which the sublists are generated
# @return a list containing all of the sublists of data
def allSublists(data):
  # Start out with the empty list as the only sublist of data
  sublists = [[]]

  # Generate all of the sublists of data from length 1 to len(data)
  for length in range(1, len(data) + 1):
    # Generate the sublists starting at each index
    for i in range(0, len(data) - length + 1):
      # Add the current sublist to the list of sublists
      sublists.append(data[i : i + length])

  # Return the result
  return sublists

# Demonstrate the allSublists function
def main():
  print("The sublists of [] are: ")
  print(allSublists([]))

  print("The sublists of [1] are: ")
  print(allSublists([1]))

  print("The sublists of [1, 2] are: ")
  print(allSublists([1, 2]))

  print("The sublists of [1, 2, 3] are: ")
  print(allSublists([1, 2, 3]))

  print("The sublists of [1, 2, 3, 4] are: ")
  print(allSublists([1, 2, 3, 4]))

# Call the main function
main()
```

A list containing an empty list is denoted by `[[]]`.

Solution to Exercise 127: The Sieve of Eratosthenes

```
##
# Determine all of the prime numbers from 2 to some limit entered
# by the user using the Sieve of Eratosthenes.
#

# Read the limit from the user
limit = int(input("Generate all primes up to what limit? "))
```

```
# Generate all of the numbers from 0 to limit
nums = []
for i in range(0, limit + 1):
  nums.append(i)

# ''Cross out'' 1 by replacing it with a 0
nums[1] = 0

# Cross out all of the multiples of each prime number that we discover
p = 2
while p < limit:
  # Cross out all multiples of p (but not p itself)
  for i in range(p*2, limit + 1, p):
    nums[i] = 0

  # Find the next number that is not crossed out
  p = p + 1
  while p < limit and nums[p] == 0:
    p = p + 1

# Display the result
print("The primes up to", limit, "are:")
for i in nums:
  if nums[i] != 0:
    print(i)
```

Dictionary Solutions

14

Solution to Exercise 128: Reverse Lookup

```
##
# Conduct a reverse lookup on a dictionary, finding all of the keys that map to the provided value.
#

## Conduct a reverse lookup on a dictionary
# @param data the dictionary to perform the reverse lookup on
# @param value the value to search for in the dictionary
# @return a list (possibly empty) of keys from data that map to value
def reverseLookup(data, value):
  # Construct a list of the keys that map to value
  keys = []

  # Check each key, adding it to keys if the values match
  for key in data:
    if data[key] == value:
      keys.append(key)

  # Return the list of keys
  return keys

# Demonstrate the reverseLookup function
def main():
  # A dictionary mapping 4 French words to their English equivalents
  frEn = {"le" : "the", "la" : "the", "livre" : "book", "pomme" : "apple"}

  # Demonstrate the reverseLookup function with 3 cases: One that returns
  # multiple keys, one that returns one key, and one that returns no keys
  print("The french words for 'the' are: ", reverseLookup(frEn, "the"))
  print("Expected: ['le', 'la']")
  print()
  print("The french word for 'apple' is: ", reverseLookup(frEn, "apple"))
  print("Expected: ['pomme']")
  print()
```

Each key in a dictionary must be unique. However, values may be repeated. As a result, performing a reverse lookup may identify zero, one or several keys that match the provided value.

B. Stephenson, *The Python Workbook*, DOI 10.1007/978-3-319-14240-1_14

```
  print("The french word for 'asdf' is: ", reverseLookup(frEn, "asdf"))
  print("Expected: []")

# Only call the main function if this file has not been imported
if __name__ == "__main__":
  main()
```

Solution to Exercise 129: Two Dice Simulation

```
##
# Simulate rolling two dice many times and compare the simulated results
# to the results expected by probability theory.
#
from random import randrange

NUM_RUNS = 1000
D_MAX = 6

## Simulate rolling two six-sided dice
# @return the total of rolling two simulated dice
def twoDice():
  # Simulate two dice
  d1 = randrange(1, D_MAX + 1)
  d2 = randrange(1, D_MAX + 1)

  # Return the total
  return d1 + d2

# Simulate many rolls and display the result
def main():
  # Create a dictionary of expected proportions
  expected = {2: 1/36, 3: 2/36, 4: 3/36, 5: 4/36, \
              6: 5/36, 7: 6/36, 8: 5/36, 9: 4/36, \
              10: 3/36, 11: 2/36, 12: 1/36}

  # Create a dictionary that maps from the total of
  # two dice to the number of occurrences
  counts = {2: 0, 3: 0, 4: 0, 5: 0, 6: 0, 7: 0, \
            8: 0, 9: 0, 10: 0, 11: 0, 12: 0}

  # Simulate NUM_RUNS rolls, and count each roll
  for i in range(NUM_RUNS):
    t = twoDice()
    counts[t] = counts[t] + 1

  # Display the simulated proportions and the expected proportions
  print("Total   Simulated   Expected")
  print("          Percent    Percent")
  for i in sorted(counts.keys()):
    print("%5d %11.2f   %8.2f" % \
          (i, counts[i] / NUM_RUNS * 100, expected[i] * 100))
```

Each dictionary is initialized so that it has keys 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 and 12. In the `expected` dictionary, the value is initialized to the probability that each key will occur when two 6-sided dice are rolled. In the `counts` dictionary, each value is initialized to 0. The values in `counts` are increased as the simulation runs.

```
# Call the main function
main()
```

Solution to Exercise 134: Unique Characters

```
##
#  Compute the number of unique characters in a string using a dictionary.
#

# Read the string from the user
s = input("Enter a string: ")

# Add each character to a dictionary with a value of True.  Once we are done the number
# of keys in the dictionary will be the number of unique characters in the string.
characters = {}
for ch in s:
  characters[ch] = True
```

> Every key in a dictionary must have a value associated with it. However, in this solution the value is never used. As a result, we have elected to associate `True` with each key, but any other value could have been used instead of `True`.

```
# Display the result
print("That string contained", len(characters), "unique character(s).")
```

> The `len` function returns the number of keys in a dictionary.

Solution to Exercise 135: Anagrams

```
##
#  Determine whether or not two strings are anagrams and report the result.
#

## Compute the frequency distribution for the characters in a string
#  @param s the string to process
#  @return a dictionary mapping each character to its count
def characterCounts(s):
  # Create a new, empty dictionary
  counts = {}

  # Update the count for each character in the string
  for ch in s:
    if ch in counts:
      counts[ch] = counts[ch] + 1
    else:
      counts[ch] = 1
```

```
  # Return the result
  return counts

# Determine if two strings entered by the user are anagrams
def main():
  # Read the strings from the user
  s1 = input("Enter the first string: ")
  s2 = input("Enter the second string: ")

  # Get the character counts for each string
  counts1 = characterCounts(s1)
  counts2 = characterCounts(s2)

  # Display the result
  if counts1 == counts2:
    print("Those strings are anagrams.")
  else:
    print("Those strings are not anagrams.")

# Call the main function
main()
```

Two dictionaries are equal if and only if both dictionaries have the same keys and for every key, *k*, the value associated with *k* is the same in both dictionaries.

Solution to Exercise 137: Scrabble™ Score

```
##
#  Use a dictionary to compute the Scrabble™ score for a word.
#

# Initialize the dictionary so that it maps from letters to points
points = {"A": 1, "B": 3, "C": 3, "D": 2, "E": 1, "F": 4, \
          "G": 2, "H": 4, "I": 1, "J": 2, "K": 5, "L": 1, \
          "M": 3, "N": 1, "O": 1, "P": 3, "Q": 10, "R": 1, \
          "S": 1, "T": 1, "U": 1, "V": 4, "W": 4, "X": 8,
          "Y": 4, "Z": 10}

# Read a word from the user
word = input("Enter a word: ")

# Compute the score for the word
uppercase = word.upper()
score = 0
for ch in uppercase:
  score = score + points[ch]

# Display the result
print(word, "is worth", score, "points.")
```

The word is converted to uppercase so that the user will get a correct response when they enter the word in upper, mixed or lowercase. This could also be accomplished by adding all of the lowercase letters to the dictionary.

Solution to Exercise 138: Create a Bingo Card

```
##
#  Create and display a random Bingo card.
#
from random import randrange

NUMS_PER_LETTER = 15

# Create a bingo card with randomly generated numbers
# @return a dictionary representing the card where the keys are the strings
#       ''B'', ''I'', ''N'', ''G'', and ''O'', and the values are lists of the
#       numbers that appear under each letter from top to bottom
def createCard():
  card = {}

  # The range of integers that can be generated for the current letter
  lower = 1
  upper = 1 + NUMS_PER_LETTER

  # For each of the five letters
  for letter in ["B", "I", "N", "G", "O"]:
    # Start with an empty list for the letter
    card[letter] = []

    # Keep generating random numbers until we have 5 unique ones
    while len(card[letter]) != 5:
      next_num = randrange(lower, upper)
      # Ensure that we do not include any duplicate numbers
      if next_num not in card[letter]:
        card[letter].append(next_num)

    # Update the range of values that will be generated for the next letter
    lower = lower + NUMS_PER_LETTER
    upper = upper + NUMS_PER_LETTER

  # Return the generated card
  return card

# Display a bingo card with nice formatting
# @param card the bingo card to display
# @return (None)
def displayCard(card):
  # Display the headings
  print("B  I  N  G  O")

  # Display the numbers on the card
  for i in range(5):
    for letter in ["B", "I", "N", "G", "O"]:
      print("%2d " % card[letter][i], end="")
    print()
```

```
# Generate a random Bingo card and display it
def main():
  card = createCard()
  displayCard(card)

# Call the main program only if this file hasn't been imported
if __name__ == "__main__":
  main()
```

File and Exception Solutions

15

Solution to Exercise 141: Display the Head of a File

```
##
# Display the head (first 10 lines) of a file whose name is provided as a command line parameter.
#
import sys

NUM_LINES = 10

# Verify that exactly one command line parameter (in addition to the .py file) was supplied
if len(sys.argv) != 2:
  print("You must provide the file name as a command line parameter.")
  quit()

try:
  # Open the file for reading
  inf = open(sys.argv[1], "r")

  # Read the first line from the file
  line = inf.readline()

  # Keep looping until we have either read and displayed 10 lines or
  # we have reached the end of the file
  count = 0
  while count < NUM_LINES and line != "":
    # Remove the trailing newline character and count the line
    line = line.rstrip()
    count = count + 1

    # Display the line
    print(line)

    # Read the next line from the file
    line = inf.readline()

  # Close the file
  inf.close()
```

When the `quit` function is called the program ends immediately.

B. Stephenson, *The Python Workbook*, DOI 10.1007/978-3-319-14240-1_15

```
except IOError:
  # Display a message if something goes wrong while accessing the file
  print("An error occurred while accessing the file.")
```

Solution to Exercise 142: Display the Tail of a File

```
##
# Display the tail (last lines) of a file whose name is provided as a command line parameter.
#
import sys

NUM_LINES = 10

# Verify that exactly one command line parameter (in addition to the .py file) was provided
if len(sys.argv) != 2:
  print("The file name must be provided as a command line parameter.")
  quit()

try:
  # Open the file for reading
  inf = open(sys.argv[1], "r")

  # Read through the file, always saving the NUM_LINES most recent lines
  lines = []
  for line in inf:
    # Add the most recent line to the end of the list
    lines.append(line)
    # If we now have more than NUM_LINES lines then remove the oldest one
    if len(lines) > NUM_LINES:
      lines.pop(0)

  # Close the file
  inf.close()

except:
  print("An error occurred while processing the file.")
  quit()

# Display the last lines of the file
for line in lines:
  print(line, end="")
```

Solution to Exercise 143: Concatenate Multiple Files

```
##
# Concatenate one or more files, displaying the result.
#
import sys
```

```
# Ensure that at least one command line parameter (in addition to the .py file) has been provided
if len(sys.argv) == 1:
  print("You must provide at least one file name.")
  quit()

# Process all of the files provided on the command line
for i in range(1, len(sys.argv)):
  fname = sys.argv[i]
  try:
    # Open the current file for reading
    inf = open(fname, "r")

    # Display the file
    for line in inf:
      print(line, end="")

    # Close the file
    inf.close()

  except:
    # Display a message, but do not quit so that the program will go on to process subsequent files
    print("Couldn't open/display", fname)
```

The element at position 0 in `sys.argv` is the Python file that is being executed. As a result, our for loop starts processing files at position 1 in the list.

Solution to Exercise 148: Sum a List of Numbers

```
##
#  Compute the sum of numbers entered by the user, ignoring non-numeric input.
#

# Read the first line of input from the user
line = input("Enter a number: ")
total = 0

# Keep reading until the user enters a blank line
while line != "":
  try:
    # Try and convert the line to a number
    num = float(line)
    # If the conversion succeeds then add it to the total and display it
    total = total + num
    print("The total is now", total)

  except ValueError:
    # Display an error message before going on to read the next value
    print("That wasn't a number.")

  # Read the next number
  line = input("Enter a number: ")

# Display the total
print("The grand total is", total)
```

Solution to Exercise 150: Remove Comments

```
##
#  Remove all of the comments from a python file, ignoring the case where
#  a comment character occurs within a string.
#

# Read and open the input file, ensuring that it was opened successfully
try:
  in_name = input("Enter the name of a Python file: ")
  inf = open(in_name, "r")
except:
  print("A problem was encountered while opening the input file.")
  print("Quitting...")
  quit()

# Read and open the output file, ensuring that it was opened successfully
try:
  out_name = input("Enter the output file name: ")
  outf = open(out_name, "w")
except:
  print("A problem was encountered while opening the output file.")
  print("Quitting...")
  quit()

try:
  # Read all of the lines from the input file, process them to remove
  # comments, and save the lines to a new file
  for line in inf:
    # Find the position of the comment character (-1 if there isn't one)
    pos = line.find("#")

    # If there is a comment then form a slice of the
    # string that excludes it, overwriting line
    if pos > -1:
      line = line[0 : pos]
      line = line + "\n"

    # Write the (potentially modified) line to the file
    outf.write(line)

  # Close the files
  inf.close()
  outf.close()

except:
  print("A problem was encountered while processing the file.")
  print("Quitting...")
```

The position of the comment character is stored in `pos`. As a result, `line[0 : pos]` is all of the characters up to but not including the comment character.

Solution to Exercise 151: Two Word Random Password

```
##
#  Generate a password by concatenating two random words.  The password will be
#  between 8 and 10 characters, and each word will be at least three letters long.
#
from random import randrange

WORD_FILE = "../Data/words.txt"

# Read all of the words from the file, only keeping those that are
# between 3 and 7 characters in length, and store them in a list.
words = []
inf = open(WORD_FILE, "r")
for line in inf:
  # Remove the newline character
  line = line.rstrip()

  # Keep words that are between 3 and 7 letters long
  if len(line) >= 3 and len(line) <= 7:
    words.append(line)

# Close the file
inf.close()

# Randomly select the first word for the password.  It can be any word.
first = words[randrange(0, len(words))]
first = first.capitalize()

# Keep selecting a second word until we find one that doesn't make the
# password too short or too long
password = first
while len(password) < 8 or len(password) > 10:
  second = words[randrange(0, len(words))]
  second = second.capitalize()
  password = first + second

# Display the generated password
print("The random password is:", password)
```

The password we are creating will be between 8 and 10 characters in length. Since the shortest acceptable word is 3 letters long, and a password must have 2 words in it, a password can never contain a word longer than 7 letters.

Solution to Exercise 153: A Book with No "e" ...

```
##
#  Determine the proportion of words that include each letter of the alphabet.
#  The letter that is used in the smallest proportion of words is highlighted.
#
WORD_FILE = "../Data/words.txt"

# Maintain a dictionary that counts the number of words containing each letter.
# Initialize the count for each letter to 0.
contains = {}
for ch in "ABCDEFGHIJKLMNOPQRSTUVWXYZ":
  contains[ch] = 0
```

```
# Open the file, and process each word, updating the contains dictionary
num_words = 0
inf = open(WORD_FILE, "r")
for word in inf:
  # Convert the word to uppercase and remove the newline character
  word = word.upper().rstrip()

  # Before we can update the dictionary we need to generate a list of the unique letters in the
  # word.  Otherwise we will increase the count multiple times for words that contain repeated
  # letters.  We also need to remove any hyphens that might be present.
  unique = []
  for ch in word:
    if ch not in unique and ch != "-":
      unique.append(ch)

  # Now increment the counts for all of the letters that are in the list of unique characters
  for ch in unique:
    contains[ch] = contains[ch] + 1

  # Keep count of the number of words that we have processed
  num_words = num_words + 1

# Close the file
inf.close()

# Display the result for each letter.  While displaying the results we will also
# determine which character had the smallest count so that we can display it later.
smallest_count = min(contains.values())
for ch in sorted(contains):
  if contains[ch] == smallest_count:
    smallest_letter = ch
  percentage = contains[ch] / num_words * 100
  print(ch, "occurs in %.2f percent of words" % percentage)

# Display the letter that is easiest to avoid based on the number of words in which it appears.
print()
print("The letter that is easiest to avoid is", smallest_letter)
```

Solution to Exercise 154: Names that Reached Number One

```
##
#  Display all of the girls' and boys' names that were the most popular in at least one year between
# 1900 and 2012.
#
FIRST_YEAR = 1900
LAST_YEAR = 2012

##
#  Load the first line from the file, extract the name, and add it to the
#  names list if it is not already present.
#  @param the name of the file to read the data from
#  @param the list to add the name to (if it isn't already present)
#  @return (None)
def LoadAndAdd(fname, names):
  # Open the file, read the first line, and extract the name
  inf = open(fname, "r")
  line = inf.readline()
  inf.close()
  parts = line.split()
  name = parts[0]

  # Add the name to the list if it is not already present
  if name not in names:
    names.append(name)

# Display the girls and boys names that reached #1 in at least one year between 1900 and 2012
def main():
  # Create two lists to store the most popular names
  girls = []
  boys = []

  # Process each year in the range, reading the first line out of the girl file and the boy file
  for year in range(FIRST_YEAR, LAST_YEAR + 1):
    girl_fname = "../Data/BabyNames/" + str(year) + "_GirlsNames.txt"
    boy_fname = "../Data/BabyNames/" + str(year) + "_BoysNames.txt"
```

My solution assumes that the baby names data files are stored in a different folder than the Python program. If you have the data files in the same folder as your program then `../Data/BabyNames/` should be omitted.

```
    LoadAndAdd(girl_fname, girls)
    LoadAndAdd(boy_fname, boys)

  # Display the lists
  print("Girls' names that reached #1:")
  for name in girls:
    print(" ", name)
  print()
```

```
  print("Boys' names that reached #1: ")
  for name in boys:
    print(" ", name)

# Call the main function
main()
```

Solution to Exercise 158: Spell Checker

```
##
#  Find and list all of the words in a file that are misspelled.
#
from only_words import onlyTheWords
from sys import argv


WORDS_FILE = "../Data/words.txt"

# Ensure that the program has the correct number of command line parameters
if len(argv) != 2:
  print("One command line parameter must be provided.  Quiting...")
  quit()

# Open the file.  Quit if the file is not opened successfully.
try:
  inf = open(argv[1], "r")
except:
  print("Failed to open '%s' for reading.  Quiting..." % argv[1])
  quit()

# Load all of the words into a dictionary of valid words.  The
# value 0 is associated with each word, but it is never used.
valid = {}
words_file = open(WORDS_FILE, "r")
for word in words_file:
  word = word.lower().rstrip()
  valid[word] = 0
words_file.close()

# Read each line from the file, adding any misspelled
# words to the list of misspellings
misspelled = []
for line in inf:
  # Discard the punctuation marks by calling the function developed in Exercise 111
  words = onlyTheWords(line)
  for word in words:
    # Only add to the misspelled list if the word is misspelled and not already in the list
    if word.lower() not in valid and word not in misspelled:
      misspelled.append(word)

# Close the file being checked
inf.close()
```

This solution uses a dictionary, but the values in the dictionary are never used. As a result, a set would be a better choice if you have learned about that data structure. A list is not used because checking if a key is in a dictionary is faster than checking if an element is in a list.

```
# Display the misspelled words, or a message indicating that no words are misspelled
if len(misspelled) == 0:
  print("No words were misspelled.")
else:
  print("The following words are misspelled:")
  for word in misspelled:
    print(" ", word)
```

Solution to Exercise 160: Redacting Text in a File

```
##
# Redact a text file by removing all occurrences of sensitive words.
# The redacted version of the text is written to a new file.
#
# Note that this program does not perform any error checking, and it
# does not implement case insensitive redaction.
#

# Get the name of the input file and open it
inf_name = input("Enter the name of the text file to redact: ")
inf = open(inf_name, "r")

# Get the name of the sensitive words file and open it
sen_name = input("Enter the name of the sensitive words file: ")
sen = open(sen_name, "r")

# Load all of the sensitive words into a list
words = []
line = sen.readline()
while line != "":
  line = line.rstrip()
  words.append(line)

  line = sen.readline()

sen.close()

# Get the name of the output file and open it
outf_name = input("Enter the name for the new redacted file: ")
outf = open(outf_name, "w")

# Read each line from the input file.  Replace all of the sensitive words
# with asterisks.  Then write the line to the output file.
line = inf.readline()
while line != "":
  # Check for and replace each sensitive word.  Use a number of asterisks that
  # matches the number of letters in the word
  for word in words:
    line = line.replace(word, "*" * len(word))
```

The sensitive word file can be closed now because all of the words have been read into a list.

```
  # Write the modified line to the output file
  outf.write(line)

  # Read the next line from the input file
  line = inf.readline()

# Close the input and output files
inf.close()
outf.close()
```

Solution to Exercise 161: Missing Comments

```
##
#  Find and display the names of Python functions that are not immediately preceded by a comment.
#
from sys import argv

# Verify that at least one file name has been provided as a command line parameter
if len(argv) == 1:
  print("At least one filename must be provided as a", \
        "command line parameter.")
  print("Quiting...")
  quit()

# Process each file provided as a command line parameter
for fname in argv[1 : len(argv)]:
  # Attempt to process the file
  try:
    inf = open(fname, "r")

    # As we move through the file we need to keep a copy of the previous
    # line so that we can check to see if it starts with a comment character.
    # We also need to keep track of the line number within the file.
    prev = " "
    lnum = 1
```

The `prev` variable must be initialized to a string that is at least one character in length. Otherwise the program will crash when the first line in the file that is being checked is a function definition.

```
    # Check each line in the current file
    for line in inf:
      # If the line is a function definition and the previous line is not a comment
      if line.startswith("def ") and prev[0] != "#":
        # Find the first ( on the line so that we know where the function name ends
        bracket_pos = line.index("(")
        name = line[4 : bracket_pos]

        # Display information about the function that is missing its comment
        print("%s line %d: %s" % (fname, lnum, name))
```

```
      # Save the current line and update the line counter
      prev = line
      lnum = lnum + 1

    # Close the current file
    inf.close()

except:
    print("A problem was encountered with file '%s'." % fname)
    print("Moving on to the next file...")
```

Recursion Solutions

16

Solution to Exercise 164: Total the Values

```
##
#  Compute the total of a collection of numbers entered by the user.  The user
#  will enter a blank line to indicate that no further numbers will be entered.
#

# Compute the total of all the numbers entered by the user until the user enters a blank line.
# @return the total of the entered values
def readAndTotal():
  # Read a value from the user
  line = input("Enter a number (blank to quit): ")

  # Base case: The user entered a blank line so the total is 0
  if line == "":
    return 0
  else:
    # Recursive case: Convert the current line to a number and use recursion to read the next line
    return float(line) + readAndTotal()

# Read a collection of numbers from the user and display the total
def main():
  # Read the values from the user and compute the total
  total = readAndTotal()

  # Display the result
  print("The total of all those values is", total)

# Call the main function
main()
```

Solution to Exercise 167: Recursive Palindrome

```
##
#  Determine whether or not a string entered by the user is a palindrome using recursion.
#
```


B. Stephenson, *The Python Workbook*, DOI 10.1007/978-3-319-14240-1_16

```
## Determine whether or not a string is a palindrome
#  @param s the string to check
#  @return True if the string is a palindrome, False otherwise
def isPalindrome(s):
  # Base case: the empty string is a palindrome.  So is a string containing only 1 character.
  if len(s) <= 1:
    return True

  # Recrusive case: The string is a palindrome only if the first and last
  # characters match, and the rest of the string is a palindrome
  return s[0] == s[len(s) - 1] and isPalindrome(s[1 : len(s) - 1])

# Check whether or not a string entered by the user is a palindrome
def main():
  # Read the input from the user
  line = input("Enter a string: ")

  # Check the status and display the result
  if isPalindrome(line):
    print("That was a palindrome!")
  else:
    print("That is not a palindrome.")

# Call the main function
main()
```

Solution to Exercise 169: String Edit Distance

```
##
# Compute and display the edit distance between two strings.
#

## Compute the edit distance between two strings as a count of the minimum number of insert,
#  delete and substitute operations needed to transform one string into the other.
#  @param s the first string
#  @param t the second string
#  @param the edit distance between the strings
def editDistance(s, t):
  # If one string is empty, then the edit distance is one insert operation
  # for each letter in the other string
  if len(s) == 0:
    return len(t)
  elif len(t) == 0:
    return len(s)
  else:
    cost = 0
    # If the last characters are not equal, set cost to 1
    if s[len(s) - 1] != t[len(t) - 1]:
      cost = 1
```

```
    # Compute three distances
    d1 = editDistance(s[0 : len(s) - 1], t) + 1
    d2 = editDistance(s, t[0 : len(t) - 1]) + 1
    d3 = editDistance(s[0 : len(s) - 1] , t[0 : len(t) - 1]) + cost

    # Return the minimum distance
    return min(d1, d2, d3)

# Compute the edit distance between two strings entered by the user
def main():
  # Read input from the user
  s1 = input("Enter a string: ")
  s2 = input("Enter another string: ")

  # Compute and display the result
  print("The edit distance between %s and %s is %d." % \
        (s1, s2, editDistance(s1, s2)))

# Call the main function
main()
```

Solution to Exercise 172: Element Sequences

```
##
# Determine the longest sequence of elements that can follow an element
# entered by the user where each element in the sequence begins with
# the same letter as the last letter of its predecessor.
#
from sys import *

ELEMENTS_FNAME = "../Data/Elements.csv"

# Find the longest sequence of words, beginning with start, where each word
# begins with the last letter of its predecessor.
# @param start the first word in the sequence
# @param the list of words that can occur in the sequence
# @return the longest list of words beginning with start that meets the
#       letter constraints outlined previously
def longestSequence(start, words):
  # Base case: If start is empty then the list of words is empty
  if start == "":
    return []

  # Find the best (longest) list of words by checking each possible word
  # that could appear next in the sequence
  best = []
  last_letter = start[len(start) - 1].lower()
  for i in range(0, len(words)):
    first_letter = words[i][0].lower()
```

```
    # If the first letter of the next word matches the last letter of the previous word
    if first_letter == last_letter:
      # Use recursion to find a candidate sequence of words
      candidate = longestSequence(words[i], \
                  words[0 : i] + words[i + 1 : len(words)])
      # Save the candidate if it is better than the best sequence that we have seen previously
      if len(candidate) > len(best):
        best = candidate

  # Return the best candidate sequence, preceeded by the starting word
  return [start] + best

# Load the names of all of the elements from the elements file
# @return a list of all the element names
def loadNames():
  names = []

  # Open the element data set
  inf = open(ELEMENTS_FNAME, "r")

  # Load each line, storing the element name in a list
  for line in inf:
    line = line.rstrip()
    parts = line.split(",")
    names.append(parts[2])

  # Close the file and return the list
  inf.close()
  return names

# Display a longest sequence of elements starting with an element entered by the user
def main():
  # Load all of the element names
  names = loadNames()

  # Read the starting element and capitalize it
  start = input("Enter the name of an element: ")
  start = start.capitalize()

  # Remove the starting element from the list of elements
  names.remove(start)

  # Find a longest sequence starting with start
  sequence = longestSequence(start, names)

  # Display the sequence of elements
  print("A longest sequence that starts with", start, "is:")
  for element in sequence:
    print(" ", element)

# Call the main function
main()
```

Solution to Exercise 174: Run-Length Encoding

```
##
#  Perform run-length encoding on a string of values using recursion.
#

## Perform run-length encoding on a string or list of values
#  @param data the string or list to encode
#  @return a list where the elements at even positions are data values and the
#          elements at odd positions are counts of the number of times that
#          the data value before it should be replicated.
def encode(data):
  # If data is empty then no encoding work is necessary
  if len(data) == 0:
    return []
```

If we performed the comparison `data == ""` then this function would only work for strings. If we performed the comparison `data == []` then it would only work for lists. Checking the length of the parameter allows the function to work for both strings and lists.

```
  # Find the index of the first item that is not the same as the item at position 0 in data
  index = 1
  while index < len(data) and data[index] == data[index - 1]:
    index = index + 1

  # Encode the current character group
  current = [data[0], index]

  # Use recursion to process the characters from index to the end of the string
  return current + encode(data[index : len(data)])

# Demonstrate the encode function
def main():
  # Read a string from the user
  s = input("Enter some characters: ")

  # Display the result
  print("When those characters are run-length encoded, the result is:")
  print(encode(s))

# Call the main function
main()
```

Index

B. Stephenson, *The Python Workbook*, DOI 10.1007/978-3-319-14240-1