2.5.1 Blackjack Example

Function Reusability

One advantage to writing functions is that once a function is written, if it's written in a general manner, like the rollPairDice function, it can be used multiple times. There are two clear benefits to using functions in this context. One benefit is that programs with functions that are reused tend to be shorter to write. Secondly, and more importantly, if we write a function and find all the errors in it, then we can call that function many times and be fairly confident that if our program has errors, the errors aren't in the function. Without the function, different errors could creep up in each "copy" of the code that essentially does the same thing.

Blackjack Program

In the following example, we will simulate a simple game of blackjack with some modifications, based on the amount of Python we currently know. In blackjack, each player is initially dealt two cards. The value of each card ranges from 2 to 11. Number cards are worth their value, face cards are worth 10 points and aces can be worth either 1 or 11 points. In this version, we'll make all aces worth 11 points to simplify the simulation. In regular blackjack, a player may deal as many extra cards as she wants. The goal of the game is to get as close to 21 points as possible, without going over. The winner is whoever gets the most points without exceeding 21. In a casino, there can be multiple winners – anyone who beats the dealer wins in a casino. But, in our version of the game, there will be two players. If both exceed 21 or both get the same score less than or equal to 21, we will declare no winner. Otherwise, whoever has the greatest score less than or equal to 21 will be the winner. Since we don't yet know how to repeat a set of statements many times, we will allow each player to only receive one extra card, if they chose.

We will employ four functions in our implementation:

1) getCards(n) – returns the sum of n cards, where n is either 1 or 2.

2) oneCard() – returns the score of one randomly selected card

3) wantCard(name, curscore) – returns True if name, who currently has curscore points wants an extra card, False otherwise.

4) printOutcome(name1, score1, name2, score2) – prints the outcome of a game between name1 and name2 where they've scored score1 and score2 points, respectively.

The most difficult part of seeing this example in its entirety is conceiving how it was designed. The first three functions return values while the last one returns nothing. Each of the four functions takes in different pieces of information. It takes time to get comfortable designing functions in programs and putting everything together. At this point, don't be concerned if you don't feel that you could design a program this complex. Designing programs that use functions effectively takes time and students gradually get better at the process. The goal of this program is to provide a more complicated example that clearly exemplifies some of the benefits of functions that aren't clear from smaller examples. (As an exercise, try rewriting this program all in main, and the benefit of functions will be crystal clear!) Download the example and run it a few times. Though an explanation will be given here, try to figure out how each function communicates with one another. There are two lines of communication: input a function receives to do its job and a return value. Functions receive input in the form of parameters. This is the name of the items inside the parentheses in a function call. When the function is done, it potentially returns a value to the function that called it. The receiving function is then responsible for using this value as it sees fit.

For example, the getCards function takes in an integer. This integer tells it how many cards to give the player. The function's job however, is to simply return the total score of these cards. The specific information about the cards themselves is actually lost. (If you run the program, you realize that you never find out which specific cards you hold.)

The oneCard function has a more simple task: It returns the value of a single random card. As you might imagine, once this function is written, we can easily write the getCards function by calling the oneCard function once or twice, as is necessary. The nature of the getCards function is general enough that we can call it for the initial deal OR to obtain the extra card. Let's look at both of these functions:

```
# Pre-condition: n must be either 1 or 2.
def getCards(n):

    # We must have at least one card.
    total = oneCard()

    # Add a second card, if necessary.
    if n == 2:
        total = total + oneCard()

    return total

# Returns a random blackjack card value, assuming
# all aces are worth 11 points.
def oneCard():

    card = random.randint(2, 14)

    # Numeric cards
    if card <= 10:
        return card

    # Face cards
    elif card <= 13:
        return 10

    # Aces!
    else:
        return 11
```

Notice that in the oneCard function, we simulate a random card by a random number in a 13 value range, to correspond to the 13 kinds of cards. Then, we assign each number to a specific kind of card. It’s easiest to assign 2 – 10 to the same values, 11 – 13 to the face cards, and 14 to an Ace. It would have been equally valid to generate a random number in between 1 and 13, inclusive and assign Aces to 1. In fact, any system where each kind has a 1/13 chance of being chosen would be valid. Once we make our selection, we calculate the appropriate number of points and return it!

In the getCards function, we call the oneCard function once, and then use an if statement to check to see if we should call it again.

Let’s take a look at the program in its entirety:

```
# Arup Guha
# 8/13/2012
# Simplified Blackjack Example with Functions

import random

def main():

    name1 = input("Player #1, what is your name?\n")
    name2 = input("Player #2, what is your name?\n")

    score1 = getCards(2)
    score2 = getCards(2)

    if wantCard(name1, score1):
        score1 = score1 + getCards(1)

    if wantCard(name2, score2):
        score2 = score2 + getCards(1)

    printOutcome(name1, score1, name2, score2)

# Pre-condition: n must be either 1 or 2.
def getCards(n):

    # We must have at least one card.
    total = oneCard()

    # Add a second card, if necessary.
    if n == 2:
        total = total + oneCard()

    return total

# Returns a random blackjack card value, assuming
# all aces are worth 11 points.
def oneCard():

    card = random.randint(2, 14)

    # Numeric cards
    if card <= 10:
        return card

    # Face cards
    elif card <= 13:
        return 10

    # Aces!
    else:
        return 11
```

```python
# Returns true iff name wants an extra card.
def wantCard(name, curscore):

    # Get the user's answer.
    print(name,", your score is currently ",curscore, sep="")
    answer = input("Would you like another card(yes/no)?\n")

    # Return accordingly.
    if answer == "yes":
        return True
    else:
        return False

# Prints the outcome of a game between name1 and name2
# where they've scored score1 and score2, respectively.
def printOutcome(name1, score1, name2, score2):

    # Print final scores.
    print(name1,", your score is ",score1,".", sep="")
    print(name2,", your score is ",score2,".", sep="")

    # Lots of cases here. Go through each.
    if score1 > 21 and score2 > 21:
        print("Sorry, no one wins, both players busted.")
    elif score2 > 21:
        print(name1,"wins because", name2, "busted.")
    elif score1 > 21:
        print(name2,"wins because", name1,"busted.")
    elif score1 > score2:
        print(name1,"wins with a higher score than",name2)
    elif score2 > score1:
        print(name2,"wins with a higher score than",name1)
    else:
        print("Both",name1,"and",name2,"tie.")


# Call main!
main()
```

2.5.2 Turtle Shapes Example

In this example, we utilize the general type of design behind the stars program (void functions which each print various types of designs), and apply it to using the Python turtle. The first function we start with is one to draw a square. We want the function to be flexible, so we need to figure what information a function who draws a square needs. We want the square to be placed in different places, and we want the square to be different sizes. So, this indicates three parameters the function needs to take in:

1) x coordinate of the lower left hand corner of the square
2) y coordinate of the lower left hand corner of the square
3) the side length of the square

Each of these values is in pixel lengths of coordinates.

When we write the function, we assume our formal parameters already have values and just think about doing the task. Here is the function definition line:

```
def drawSquare(x, y, sidelen):
```

In words, here is what we must do:

1) Move to (x, y) without drawing anything.
2) Change our heading to be in the positive x-axis.
3) Repeat four times:
    a) Move forward sidelen pixels
    b) Turn left 90 degrees

1) To move without drawing, we need to lift the pen up then use setpos.

2) To change our heading, we first need to find where we are currently heading and then correct it by turning the opposite direction the appropriate amount.

We've done #3 before. Here is the full function:

```
# Draws a square with the bottom left corner at (x, y) with a side length
# of sidelen.
def drawSquare(x, y, sidelen):

    # Lift the pen and move to the bottom right corner and put the pen down.
    turtle.penup()
    turtle.setpos(x, y)
    turtle.pendown()

    # Find how far off we are (angle wise) from heading in the +x axis.
    # This turn back so we are facing the +x axis.
    curheading = turtle.heading()
    turtle.right(curheading)
    # Draw square.
    for i in range(4):
        turtle.forward(sidelen)
        turtle.left(90)
```

Similarly, we can draw a triangle:

```
# Draws a equilateral triangle with the bottom left corner at (x, y)
# with a side length of sidelen.
def drawTriangle(x, y, sidelen):

    # Lift the pen and move to the bottom right corner and put the
    # pen down.
    turtle.penup()
    turtle.setpos(x, y)
    turtle.pendown()

    # Find how far off we are (angle wise) from heading in the +x
    # axis. This turn back so we are facing the +x axis.
    curheading = turtle.heading()
    turtle.right(curheading)

    # Draw triangle.
    for i in range(3):
        turtle.forward(sidelen)
        turtle.left(120)
```

Now, consider using these tools to build a house!!! Here is what one class came up with:

```
# Draws a house with the bottom left corner at (x, y), with a square
# and triangle with side length sidelen, and some windows!
def house(x,y,sidelen):

    # Main frame of house.
    drawSquare(x, y, sidelen)
    drawTriangle(x, y+sidelen, sidelen)

    # Windows and door
    drawSquare(x+sidelen//8, y+5*sidelen//8, sidelen//4)
    drawSquare(x+5*sidelen//8, y+5*sidelen//8, sidelen//4)
    drawSquare(x+3*sidelen//8, y, sidelen//4)
    drawSquare(x+3*sidelen//8, y+sidelen//4, sidelen//4)

    # Top window
    drawSquare(x+3*sidelen//8, y+9*sidelen//8, sidelen//4)

    # Door knob
    turtle.penup()
    turtle.setpos(x+3*sidelen//8, y+sidelen//4-5)
    turtle.pendown()
    turtle.circle(5)
```

One thing that you might notice is that drawSquare and drawTriangle are extremely similar. In fact, they almost seem like the same function. What if we wrote a function that could draw any regular polygon with 3 or more sides? This function would just have to take in the number of sides, in addition to a corner of the polygon and the side length. The key issue is calculating how much to turn each time. For a triangle, we turn 120 degrees and for a square we turn 90 degrees. If we work out some cases on paper, we see that for a pentagon we would turn 72 degrees and for a hexagon we would turn 60 degrees. Notice this following pattern:

Triangle	3 sides	120 degrees,	3 x 120 = 360
Square	4 sides	90 degrees,	4 x 90 = 360
Pentagon	5 sides	72 degrees,	5 x 72 = 360
Hexagon	6 sides	60 degrees,	6 x 60 = 260

Now, let's see if this is a coincidence or not. Recall that after we are done drawing the polygon, we will have spun around the whole circle. Since the whole circle is 360 degrees, and we make exactly n turns between sides, since each turn is equal, we MUST MAKE a turn of 360/n degrees, in order to return back to the same heading after drawing the polygon. It follows that this isn't a coincidence, but rather a genuine pattern with a proof for the general formula. Thus, the correct angle to turn is 360/n, where n is the number of sides. Now let's look at our draw polygon function:

```
def drawPolygon(n, x, y, sidelen):

    # Lift the pen and move to the bottom right corner and put the pen
    # down.
    turtle.penup()
    turtle.setpos(x, y)
    turtle.pendown()

    # Find how far off we are (angle wise) from heading in the +x
    # axis. This turn back so we are facing the +x axis.
    curheading = turtle.heading()
    turtle.right(curheading)

    # Draw the polygon
    for i in range(n):
        turtle.forward(sidelen)
        turtle.left(360/n)
```