4.3 Parameters and Arguments

Functions represent processes. A function is a named block of code that can be invoked from any point in the program. Much of the power of functions comes from our ability to supply inputs to these processes. The inputs to a function are called its *arguments* and they are represented inside the function definition by variables called *parameters*. In this section we will describe parameters and their relationships to arguments.

As we said in Section 4.2, the parameters are listed inside the parentheses in the function header. For example, we might have the following header to a function:

```
def foo(x, y):
```

This is the start of the definition of a function named foo. The parameters for this function are named x and y. These are variables, just like any other variables used in the function with one exception: variables x and y are given initial values when the function is called. The *call* of foo has the form

```
foo(<first_arg>, <second_arg>)
```

When this is executed < first_arg > is evaluated and its value becomes the starting value for parameter x. Similarly, <second_arg> is evaluated for the initial value of y. The body of foo is then executed until it ends or returns.

As a first example, we will write a function to calculate a 20% tip to add to the bill for a meal. To keep our arithmetic simple at mealtime, in cases where the tip is not an exact dollar amount we will round up to the next dollar. The tip calculation is easy: we get 20% of a value by multiplying by 0.2, and we round up with the ceiling function from the math library (see Section 2.4). For a given cost we compute the tip as ceil(cost*0.2)l our function returns this value:

```
def Tip(mealCost):
    tipAmount = ceil(mealCost*0.2)
    return tipAmount
```

Here is a full program that makes use of this:

```
def Tip(mealCost):
    tipAmount = ceil(mealCost*0.2)
    return tipAmount

def main():
    done = False
    while not done:
        amount=eval(input("Enter a meal cost, or 0 to exit: "))
        if amount == 0:
            done = True
        else:
            print( "Your tip should be $%d."%Tip(amount) )
            print( "The total meal cost is $%.2f."%
                        (amount+Tip(amount)))

main()
```

Program 4.3.1: Computing a tip

Note that the argument to function Tip() has a different name than the function's parameter: the argument is called amount while the parameter is mealCost. This is typical; we name the function's parameter something that is internally descriptive within the function. The argument can be anything we wish to apply the function to. At the time of the call the argument is evaluated and it is this value that is given to the function as an initial value for the parameter; any name present in the argument is irrelevant once its value is retrieved. For example, the following calls to function Tip are all valid:

- Tip(23)
- Tip(3*5+8)
- x = 23
  Tip(x)

It is generally not a good idea to give the parameters of a function the same names as variables elsewhere in the program, since this gives the impression that these are the same variable. The next program prints 2 as the value of myVariable, even though function Change() seems to change this value to 33.

```
def Change(myVariable):
    myVariable = 33

def main():
    myVariable = 2
```

```
    Change(myVariable)
    print( myVariable )

main()
```

What is happening here is that function Change() and function main() each have their own variables called myVariable and these variables are unrelated; a change to one does not affect the other.

If a function has multiple parameters, the arguments in a call to the function must match the parameters: there must be one argument for each parameter and they must appear in the same order. For example, function PrintMultiples () in the next program has two parameters: the first is a string and the second an integer. We could call this with PrintMultiples ("bob", 5) to print the string "bob bob bob bob bob ", but we couldn't call it with PrintMultiples (5, "bob") or with PrintMultiples ("bob").

```
def PrintMultiples(string, count):
    print( (string+" ")*count )

def main():
    PrintMultiples( "bob", 5)

main()
```

A parameter is just like any other variable of a function, with the one exception that it gets an initial value from the argument at the time the function is called. Like all other variables it can be modified within the function, and like all other variables it is invisible outside the function. It might be tempting to write something like the following, but the function call SetTo23(x) does not change the value of x, in spite of its name.

```
def SetTo23(x):
    x = 23

def main():
    x = 5
    SetTo23(x)
    print x

main()
```

Function SetTo23() changes its own variable x and has no affect on the variable x inside function main(). This program prints 5, not 23.

Examples

We will now give several examples of the way programs can be designed with functions.

First, think back to the prime number programs we wrote in Section 3.4. Each of these contained a block of code that determined whether a given number was prime. This task is clearer if it is performed by a function such as the following:

```
def IsPrime(number):
    # This returns True if number is prime,
    # and False otherwise
    for factor in range(2, number):
        if number % factor == 0:
            return False
    return True
```

Note that this function returns a Boolean value that just says if its argument is prime or not.

We can put this into a program that checks all the numbers between 2 and some upper limit and prints the primes. The result is Program 4.3.2:

```
# This prints all of the prime numbers up to
# a limit supplied by the user

def IsPrime(number):
    # This returns True if number is prime,
    # and False otherwise
    for factor in range(2, number):
        if number % factor == 0:
            return False
    return True

def main():
    max = eval(input("Enter the largest number of check: "))
    for x in range(2, max+1):
        if IsPrime(x):
            print( "%d is prime." % x )

main()
```

Program 4.3.2: Improved version of Program 3.4.4

Note how much easier to read this program is than Program 3.4.4. The main() function consists of an **input**-statement, and a loop with a conditional **print**-statement. The condition on the **print**-statement is almost grammatical English: " if IsPrime(x)" is easy to understand. In general, the more natural language we can work into a program the easier it is to read, and the more likely we are to write it correctly in the first place.

Similarly, here is a version of Program 3.4.5, which prints a table of the first N prime numbers. It is tempting to think of adding another function to handle the printing, but this is more complex than it might seem. All of the variables of a function go away after the function call, so a function could not easily keep track of the number of values that are on the current line. One way to handle that might be to build up a list of all of the values for the current line and send them to a function to print; that will need to wait until we have a more thorough treatment of lists in Chapter 5.

```
# This prints the first N prime numbers,
# where the value of N is supplied by the user.
# The output is printed in C columns.

def IsPrime(number):
    # This returns True if number is prime,
    # and False otherwise
    for factor in range(2, number):
        if number % factor == 0:
                return False
    return True

def main():
    N = eval(input("How many prime numbers do you want? "))
    C = eval(input("How many columns of output do you want? "))
    x = 2
    primeCount = 0
    lineCount = 0
    while primeCount < N:
        if IsPrime(x):
            print( "%7d " % x, end='''' )
            primeCount = primeCount + 1
            lineCount = lineCount + 1
            if lineCount == C:
                print( )
                lineCount = 0

        x = x + 1
    if lineCount != 0:
        print( )

main()
```

Program 4.3.3: Improved version of Program 3.4.5

Here is a third example with prime numbers. This time we will find *twin primes*: pairs of consecutive odd numbers that are both prime, such as 11 and 13. There is an unproven conjecture in Mathematics claiming that there are infinitely many twin prime pairs. Our program would require some contorted code if we only used loops, but with our IsPrime() function it is quite simple. Our condition for printing a pair (x, x+2) is just that both x and x+2 are prime:
if IsPrime(x) **and** IsPrime(x+2)

```
# This program looks through the numbers up to a limit
# for pairs of twin primes.

def IsPrime(number):
    # This returns True if number is prime,
    # and False otherwise
    for factor in range(2, number):
        if number % factor == 0:
                return False
    return True

def main():
    print( "This program looks for twin primes." )
    max = eval(input("Enter the largest number of check: "))
    for x in range(3, max+1):
        if IsPrime(x) and IsPrime(x+2):
            print("(%d, %d) are twin primes." % (x, x+2))

main()
```

Program 4.3.4: Finding twin primes

Note that the function IsPrime() is identical in each of these programs. Because we write functions as stand-alone objects in programs, function definitions can frequently be copied from one program to another. Once a function is written and debugged, you can use it in any programming situation where it is appropriate. This makes the job of the programmer easier, and makes our programs much more likely to be correct.

As a final example for this section we will write a program that inputs strings and determines whether they are palindromes the same forwards as backwards, such as "bob" or "racecar". To make this more interesting, palindromists usually ignore white space, punctuation and capitalization. Probably the most famous palindrome in English is "A man, a plan, a canal: Panama." Here are some others: "Dammit, I'm mad!", "Step on no pets.", "Was it a rat I saw?" and "Are we not drawn onward, we few, drawn onward to new era?"

Our strategy in developing this program is to write it one step at a time.

Whenever we need a new activity in the program we will create a function to handle it, first writing the call to the function and then writing the function itself. The steps are:

a. A loop that prompts the user for input, reads strings, and decides if each string is a palindrome. This should also print the result for the user.

b. To determine whether a string is a palindrome, we can reverse it and compare that to the original.

c. To handle punctuation and white space, we can walk through the letters of the string and remove anything that isn't a letter. Since strings are immutable, we build up a new string consisting of just the letters of the original.

d. To handle capitalization, as we build up the new string we insert the lower-case form of each letter. The string method lower() is useful for this: if s is a string, s.lower() is a string with the same letters as s, all converted to lower-case.

The first step is a straightforward application of the input loop model from Section 3.7:

```
def main():
    done = False
    while not done:
        phrase = input( "Enter a string: " )
        if phrase == "":
            done = True
        else:
            if <the phrase is a palindrome>:
                print( "'%s' is a palindrome."%
                       phrase)
            else:
                print( "'%s' is not a palindrome."%
                       phrase )
```

Palindromes - First Version

This has pseudo-code for the condition on the print statement:

```
if <the phrase is a palindrome>:
```

We can define a function to handle this check. To keep its name similar to its usage in English, we will call the function IsPalindrome(). Its job is to determine whether the variable phrase holds a palindrome; phrase is the only input it needs. This means we can state our condition as

```
if IsPalindrome(phrase):
```

Our function definition needs a parameter as a placeholder for the argument. Since this is a string we will just use the letter s:

```
def IsPalindrome(s):
```

To conclude the first step, we will make this function return True for all strings. This allows us to make sure the initial part of our program is working correctly. The following program isn't complete, but it is a working program. It allows the user to enter strings until a blank string is given. It claims that each string is a palindrome.

```
def IsPalindrome(s):
    # This returns True if string s is a palindrome
    # and False if it is not.
    return True

def main():
    done = False
    while not done:
        phrase = input( "Enter a string: " )
        if phrase == "":
            done = True
        else:
            if IsPalindrome(phrase):
                print("'%s' is a palindrome."%phrase)
            else:
                print("'%s' is not a palindrome."%phrase)

main()
```

Palindromes continued

Our next step is to make the IsPalindrome() function more useful. We defined a palindrome as a string that reads the same forwards and backwards. This is easy to code into a function:

```
def IsPalindrome(s):
    if s == Reverse(s):
        return True
    else:
        return False
```

This involves another function: Reverse(s) is a function we will write that returns the reversal of string vars. How do we do that? It is easy to use a **for**-loop to walk through the letters of a string. We want to build up a new string, which we will call answer, that is the reversal of s. Variable answer starts off as the

empty string. At each step we add the new letter onto answer. If we add it onto the end of answer, we will make an exact copy of string s. If we add the new letter onto the beginning of answer, we get the reversal of string s. If you don't see this, note that the last letter of string s will become the first letter of the answer, the next-to-last letter of s will be the second letter of the answer, and so forth.

```
def Reverse(s):
    answer = ""
    for c in s:
        answer = c + answer
    return answer
```

Adding this to our program gives the first version of our palindrome checker:

```
def Reverse(s):
    # This returns the reversal of string s:
    # if s is 'abc' this returns 'cba'.
    answer = ""
    for c in s:
        answer = c + answer
    return answer

def IsPalindrome(s):
    # This returns True if string s is a palindrome
    # and False if it is not.
    s = StripPunctuation(s)
    if s == Reverse(s):
        return True
    else:
        return False

def main():
    done = False
    while not done:
        phrase = input( "Enter a string: " )
        if phrase == "":
            done = True
        else:
            if IsPalindrome(phrase):
                print("'%s' is a palindrome."%phrase)
            else:
                print("'%s' is not a palindrome."%phrase)

main()
```

Palindromes Step 2

For the last steps, we need to eliminate the punctuation. A new function StripPunctuation() will handle that; this function will take a string s and return a new string consisting of just the alphabetic letters of s, all in lowercase. We should think carefully about where to call this function. We could call it in main(), as soon as we read the string and see that it is not empty:

```
phrase = input( "Enter a string: " )
if phrase == "":
        done = True
else:
        phrase = StripPunctuation(phrase)
        if IsPalindrome(phrase):
```

⋮

This is not good. The user will enter a phrase such as "Dammitit I'm mad!" and the program will respond that `'dammitimmad' is a palindrome.` This is confusing because 'dammitimmad' is not what the user entered. A more appropriate placement for the call to StripPalindrome() is within the IsPalindrome() function. It is this function that is trying to decide if a string is a palindrome. Anything it does to the string will be invisible to the main() function. Accordingly, we will keep main() as it stands and change the IsPalindrome() function to:

```
def IsPalindrome(s):
    s = StripPunctuation(s)
    if s == Reverse(s):
        return True
    else:
        return False
```

All that remains is to write StripPunctuation(). As with Reverse(), this walks through the letters of string s and places them in an answer string. This time we add the new letters onto the end of answer, so that we are making a copy of s. We add only the alphabetic letters to delete the punctuation and white space elements. How do we tell which are the alphabetic characters? One way to test of variable c contains a lower-case letter is to determine if it is between 'a' and 'z':

```
if 'a' <= c and c <= 'z':
```

We could do the same thing with upper-case letters and get code that starts

```
if ('a' <= c and c <= 'z) or ('A' <= c and c <= 'Z'):
```

Another way is to use a string method that tests if all of the elements of the string are alphabetic::

```
if c.isalpha():
```

Finally, we use the string method lower() to insure that we add the lower-case version of each letter to answer. Here is the code for function StripPunctuation():

```
def StripPunctuation(s):
    answer = ""
    for c in s:
        if c.isalpha():
            answer = answer + c.lower()
    return answer
```

This completes the program. Here is complete code for our palindrome program, including comments to make it more readable:

```
# This program reads strings from the user and
# says if they are palindromes: the same when
# read backwards as when read forwards

def StripPunctuation(s):
    # This returns a string just like s only all
    # of the non-letters are removed and the letters
    # are all changed to lower-case.
    answer = ""
    for c in s:
        if c.isalpha():
            answer = answer + c.lower()
    return answer

def Reverse(s):
    # This returns the reversal of string s:
    # if s is 'abc' this returns 'cba'.
    answer = ""
    for c in s:
        answer = c + answer
    return answer

def IsPalindrome(s):
    # This returns True if string s is a palindrome
    # and False if it is not.
    s = StripPunctuation(s)
    if s == Reverse(s):
        return True
    else:
        return False

def main():
    done = False
    while not done:
        phrase = input( "Enter a string: " )
        if phrase == "":
            done = True
        else:
            if IsPalindrome(phrase):
                print(" '%s' is a palindrome." % phrase)
            else:
                print(" '%s' is not a palindrome." % phrase)

main()
```

Program 4.3.5: Palindromes: final version