

Python Programming: An Introduction to Computer Science



Chapter 6 Defining Functions

Python Programming, 1/e

1

Objectives

- To understand why programmers divide programs up into sets of cooperating functions.
- To be able to define new functions in Python.
- To understand the details of function calls and parameter passing in Python.

Python Programming, 1/e

2

Objectives (cont.)

- To write programs that use functions to reduce code duplication and increase program modularity.

Python Programming, 1/e

3

The Function of Functions

- So far, we've seen four different types of functions:
 - Our programs comprise a single function called `main()`.
 - Built-in Python functions (`abs`)
 - Functions from the standard libraries (`math.sqrt`)
 - Functions from the graphics module (`p.getX()`)

Python Programming, 1/e

4

The Function of Functions

- In the code on page 166, the code for drawing the bars occurs in two different places.
 - The first bar is drawn immediately before the loop.
 - The remaining bars are drawn within the loop.

Python Programming, 1/e

5

The Function of Functions

- Having similar or identical code in more than one place has some drawbacks.
 - Issue one: writing the same code twice or more.
 - Issue two: This same code must be maintained in two separate places.
- Functions can be used to reduce code duplication and make programs more easily understood and maintained.

Python Programming, 1/e

6

Functions, Informally

- A function is like a *subprogram*, a small program inside of a program.
- The basic idea – we write a sequence of statements and then give that sequence a name. We can then execute this sequence at any time by referring to the name.

Python Programming, 1/e

7

Functions, Informally

- The part of the program that creates a function is called a *function definition*.
- When the function is used in a program, we say the definition is *called* or *invoked*.

Python Programming, 1/e

8

Functions, Informally

- Happy Birthday lyrics...

```
def main():
    print "Happy birthday to you!"
    print "Happy birthday to you!"
    print "Happy birthday, dear Fred..."
    print "Happy birthday to you!"
```

- Gives us this...

```
>>> main()
Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Fred...
Happy birthday to you!
```

Python Programming, 1/e

9

Functions, Informally

- There's some duplicated code in the program! (`print "Happy birthday to you!"`)
- We can define a function to print out this line:
- With this function, we can rewrite our program.

```
def happy():
    print "Happy birthday to you!"
```

Python Programming, 1/e

10

Functions, Informally

- The new program –

```
def singFred():
    happy()
    happy()
    print "Happy birthday, dear Fred..."
    happy()
```

- Gives us this output –

```
>>> singFred()
Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Fred...
Happy birthday to you!
```

Python Programming, 1/e

11

Functions, Informally

- Creating this function saved us a lot of typing!
- What if it's Lucy's birthday? We could write a new `singLucy` function!

```
def singLucy():
    happy()
    happy()
    print "Happy birthday, dear Lucy..."
    happy()
```

Python Programming, 1/e

12

Functions, Informally

- We could write a main program to sing to both Lucy and Fred

```
def main():
    singFred()
    print
    singLucy()
```

- This gives us this new output

```
>>> main()
Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Fred..
Happy birthday to you!

Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Lucy...
Happy birthday to you!
```

Python Programming, 1/e

13

Functions, Informally

- This is working great! But... there's still a lot of code duplication.
- The only difference between `singFred` and `singLucy` is the name in the third `print` statement.
- These two routines could be collapsed together by using a *parameter*.

Python Programming, 1/e

14

Functions, Informally

- The generic function *sing*

```
def sing(person):
    happy()
    happy()
    print "Happy birthday, dear", person + "."
    happy()
```

- This function uses a parameter named *person*. A *parameter* is a variable that is initialized when the function is called.

Python Programming, 1/e

15

Functions, Informally

- Our new output –

```
>>> sing("Fred")
Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Fred.
Happy birthday to you!
```

- We can put together a new main program!

Python Programming, 1/e

16

Functions, Informally

- Our new main program:

```
def main():
    sing("Fred")
    print
    sing("Lucy")
```

- Gives us this output:

```
>>> main()
Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Fred.
Happy birthday to you!

Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Lucy.
Happy birthday to you!
```

Python Programming, 1/e

17

Future Value with a Function

- In the future value graphing program, we see similar code twice:

```
# Draw bar for initial principal
bar = Rectangle(Point(0, 0), Point(1, principal))
bar.setFill("green")
bar.setWidth(2)
bar.draw(win)
```

```
bar = Rectangle(Point(year, 0), Point(year+1, principal))
bar.setFill("green")
bar.setWidth(2)
bar.draw(win)
```

Python Programming, 1/e

18

Future Value with a Function

- To properly draw the bars, we need three pieces of information.
 - The year the bar is for
 - How tall the bar should be
 - The window the bar will be drawn in
- These three values can be supplied as parameters to the function.

Python Programming, 1/e

19

Future Value with a Function

- The resulting function looks like this:

```
def drawBar(window, year, height):  
    # Draw a bar in window starting at year with given height  
    bar = Rectangle(Point(year, 0), Point(year+1, height))  
    bar.setFill("green")  
    bar.setWidth(2)  
    bar.draw(window)
```
- To use this function, we supply the three values. If win is a Graphwin, we can draw a bar for year 0 and principal of \$2000 using this call:

```
drawBar(win, 0, 2000)
```

Python Programming, 1/e

20

Functions and Parameters: The Details

- It makes sense to include the year and the principal in the drawBar function, but why send the window variable?
- The *scope* of a variable refers to the places in a program a given variable can be referenced.

Python Programming, 1/e

21

Functions and Parameters: The Details

- Each function is its own little subprogram. The variables used inside of one function are *local* to that function, even if they happen to have the same name as variables that appear inside of another function.
- The only way for a function to see a variable from another function is for that variable to be passed as a parameter.

Python Programming, 1/e

22

Functions and Parameters: The Details

- Since the GraphWin in the variable win is created inside of main, it is not directly accessible in drawBar.
- The window parameter in drawBar gets assigned the value of win from main when drawBar is called.

Python Programming, 1/e

23

Functions and Parameters: The Details

- A function definition looks like this:

```
def <name>(<formal-parameters>):  
    <body>
```
- The name of the function must be an identifier
- Formal-parameters is a possibly empty list of variable names

Python Programming, 1/e

24

Functions and Parameters: The Details

- Formal parameters, like all variables used in the function, are only accessible in the body of the function. Variables with identical names elsewhere in the program are distinct from the formal parameters and variables inside of the function body.

Python Programming, 1/e

25

Functions and Parameters: The Details

- A function is called by using its name followed by a list of *actual parameters* or *arguments*.
<name>(<actual-parameters>)
- When Python comes to a function call, it initiates a four-step process.

Python Programming, 1/e

26

Functions and Parameters: The Details

- The calling program suspends execution at the point of the call.
- The formal parameters of the function get assigned the values supplied by the actual parameters in the call.
- The body of the function is executed.
- Control returns to the point just after where the function was called.

Python Programming, 1/e

27

Functions and Parameters: The Details

- Let's trace through the following code:

```
def sing(person):  
    print  
    sing("Lucy")
```
- When Python gets to `def sing("Fred")`, execution of `main` is temporarily suspended.
- Python looks up the definition of `sing` and sees that it has one formal parameter, `person`.

Python Programming, 1/e

28

Functions and Parameters: The Detail

- The formal parameter is assigned the value of the actual parameter. It's as if the following statement had been executed:

```
person = "Fred"
```

Python Programming, 1/e

29

Functions and Parameters: The Details

```
def main():  
    person = "Fred"  
    sing("Fred")  
    print  
    sing("Lucy")  
  
def sing(person):  
    happy()  
    happy()  
    print "Happy birthday, dear", person + ".  
    happy()  
  
person: "Fred"
```

Note that the variable `person` has just been initialized.

Python Programming, 1/e

30

Functions and Parameters: The Details

- At this point, Python begins executing the body of `sing`.
- The first statement is another function call, to `happy`. What happens next?
- Python suspends the execution of `sing` and transfers control to `happy`.
- `happy` consists of a single `print`, which is executed and control returns to where it left off in `sing`.

Python Programming, 1/e

31

Functions and Parameters: The Details

```
def main():
    person = "Fred"
    sing("Fred")
    print
    sing("Lucy")

def sing(person):
    happy()
    happy()
    print "Happy birthday, dear", person + " ."
    happy()

def happy():
    print "Happy Birthday to you!"

person: "Fred"
```

- Execution continues in this way with two more trips to `happy`.
- When Python gets to the end of `sing`, control returns to `main` and continues immediately following the function call.

Python Programming, 1/e

32

Functions and Parameters: The Details

```
def main():
    person = "Fred"
    sing("Fred")
    print
    sing("Lucy")

def sing(person):
    happy()
    happy()
    print "Happy birthday, dear", person + " ."
    happy()
```

- Notice that the `person` variable in `sing` has disappeared!
- The memory occupied by local function variables is reclaimed when the function exits.
- Local variables do **not** retain any values from one function execution to the next.

Python Programming, 1/e

33

Functions and Parameters: The Details

- The next statement is the bare `print`, which produces a blank line.
- Python encounters another call to `sing`, and control transfers to the `sing` function, with the formal parameter "Lucy".

Python Programming, 1/e

34

Functions and Parameters: The Details

```
def main():
    sing("Fred")
    print
    sing("Lucy")

def sing(person):
    happy()
    happy()
    print "Happy birthday, dear", person + " ."
    happy()

person: "Lucy"
```

- The body of `sing` is executed for Lucy with its three side trips to `happy` and control returns to `main`.

Python Programming, 1/e

35

Functions and Parameters: The Details

```
def main():
    sing("Fred")
    print
    sing("Lucy")

def sing(person):
    happy()
    happy()
    print "Happy birthday, dear", person + " ."
    happy()
```

Python Programming, 1/e

36

Functions and Parameters: The Details

- One thing not addressed in this example was multiple parameters. In this case the formal and actual parameters are matched up based on *position*, e.g. the first actual parameter is assigned to the first formal parameter, the second actual parameter is assigned to the second formal parameter, etc.

Python Programming, 1/e

37

Functions and Parameters: The Details

- As an example, consider the call to `drawBar`:
`drawBar(win, 0, principal)`
- When control is passed to `drawBar`, these parameters are matched up to the formal parameters in the function heading:

```
def drawBar(window, year, height):
```

Python Programming, 1/e

38

Functions and Parameters: The Details

- The net effect is as if the function body had been prefaced with three assignment statements:

```
window = win
year = 0
height = principal
```

Python Programming, 1/e

39

Getting Results from a Function

- Passing parameters provides a mechanism for initializing the variables in a function.
- Parameters act as inputs to a function.
- We can call a function many times and get different results by changing its parameters.

Python Programming, 1/e

40

Functions That Return Values

- We've already seen numerous examples of functions that return values to the caller.
- `discRt = math.sqrt(b*b - 4*a*c)`
- The value `b*b - 4*a*c` is the actual parameter of `math.sqrt`.
- We say `sqrt` *returns* the square root of its argument.

Python Programming, 1/e

41

Functions That Return Values

- This function returns the square of a number:

```
def square(x):
    return x*x
```
- When Python encounters `return`, it exits the function and returns control to the point where the function was called.
- In addition, the value(s) provided in the `return` statement are sent back to the caller as an expression result.

Python Programming, 1/e

42

Functions That Return Values

- `>>> square(3)`
9
- `>>> print square(4)`
16
- `>>> x = 5`
`>>> y = square(x)`
`>>> print y`
25
- `>>> print square(x) + square(3)`
34

Python Programming, 1/e

43

Functions That Return Values

- We can use the square function to write a routine to calculate the distance between (x_1, y_1) and (x_2, y_2) .

```
def distance(p1, p2):  
    dist = math.sqrt(square(p2.getX() - p1.getX()) +  
                    square(p2.getY() - p1.getY()))  
    return dist
```

Python Programming, 1/e

44

Functions That Return Values

- Sometimes a function needs to return more than one value.
- To do this, simply list more than one expression in the `return` statement.
- ```
def sumDiff(x, y):
 sum = x + y
 diff = x - y
 return sum, diff
```

Python Programming, 1/e

45

## Functions That Return Values

- When calling this function, use simultaneous assignment.
- ```
num1, num2 = input("Please enter two numbers (num1, num2) ")  
s, d = sumDiff(num1, num2)  
print "The sum is", s, "and the difference is", d
```
- As before, the values are assigned based on position, so *s* gets the first value returned (the sum), and *d* gets the second (the difference).

Python Programming, 1/e

46

Functions That Return Values

- One “gotcha” – all Python functions return a value, whether they contain a `return` statement or not. Functions without a `return` hand back a special object, denoted `None`.
- A common problem is writing a value-returning function and omitting the `return`!

Python Programming, 1/e

47

Functions That Return Values

- If your value-returning functions produce strange messages, check to make sure you remembered to include the `return`!

Python Programming, 1/e

48

Functions that Modify Parameters

- Return values are the main way to send information from a function back to the caller.
- Sometimes, we can communicate back to the caller by making changes to the function parameters.
- Understanding when and how this is possible requires the mastery of some subtle details about how assignment works and the relationship between actual and formal parameters.

Python Programming, 1/e

49

Functions that Modify Parameters

- Suppose you are writing a program that manages bank accounts. One function we would need to do is to accumulate interest on the account. Let's look at a first-cut at the function.

```
def addInterest(balance, rate):  
    newBalance = balance * (1 + rate)  
    balance = newBalance
```

Python Programming, 1/e

50

Functions that Modify Parameters

- The intent is to set the balance of the account to a new value that includes the interest amount.
- Let's write a main program to test this:

```
def test():  
    amount = 1000  
    rate = 0.05  
    addInterest(amount, rate)  
    print amount
```

Python Programming, 1/e

51

Functions that Modify Parameters

- We hope that that the 5% will be added to the amount, returning 1050.
- >>> test()
1000
- What went wrong? Nothing!

Python Programming, 1/e

52

Functions that Modify Parameters

- The first two lines of the test function create two local variables called `amount` and `rate` which are given the initial values of 1000 and 0.05, respectively.

```
def addInterest(balance, rate):  
    newBalance = balance * (1 + rate)  
    balance = newBalance  
  
def test():  
    amount = 1000  
    rate = 0.05  
    addInterest(amount, rate)  
    print amount
```

Python Programming, 1/e

53

Functions that Modify Parameters

- Control then transfers to the `addInterest` function.
- The formal parameters `balance` and `rate` are assigned the values of the actual parameters `amount` and `rate`.
- Even though `rate` appears in both, they are separate variables (because of scope rules).

```
def addInterest(balance, rate):  
    newBalance = balance * (1 + rate)  
    balance = newBalance  
  
def test():  
    amount = 1000  
    rate = 0.05  
    addInterest(amount, rate)  
    print amount
```

Python Programming, 1/e

54

Functions that Modify Parameters

- The assignment of the parameters `balance` and `rate` in `addInterest` causes the variables `balance` and `rate` in `addInterest` to refer to the *values* of the actual parameters!

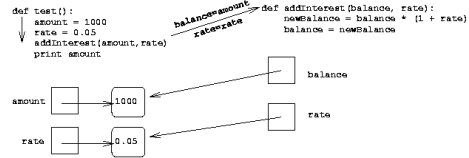
```
def addInterest(balance, rate):
    newBalance = balance * (1 +
rate)
    balance = newBalance

def test():
    amount = 1000
    rate = 0.05
    addInterest(amount, rate)
    print amount
```

Python Programming, 1/e

55

Functions that Modify Parameters



Python Programming, 1/e

56

Functions that Modify Parameters

- Executing the first line of `addInterest` creates a new variable, `newBalance`.
- `balance` is then assigned the value of `newBalance`.

```
def addInterest(balance, rate):
    newBalance = balance * (1 +
rate)
    balance = newBalance

def test():
    amount = 1000
    rate = 0.05
    addInterest(amount, rate)
    print amount
```

Python Programming, 1/e

57

Functions that Modify Parameters

- `balance` now refers to the same value as `newBalance`, but this had no effect on `amount` in the `test` function.

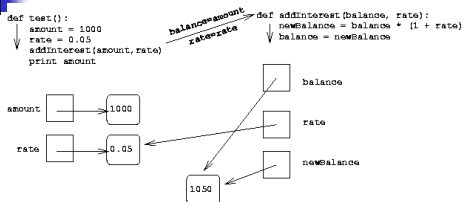
```
def addInterest(balance, rate):
    newBalance = balance * (1 +
rate)
    balance = newBalance

def test():
    amount = 1000
    rate = 0.05
    addInterest(amount, rate)
    print amount
```

Python Programming, 1/e

58

Functions that Modify Parameters



Python Programming, 1/e

59

Functions that Modify Parameters

- Execution of `addInterest` has completed and control returns to `test`.
- The local variables, including the parameters, in `addInterest` go away, but `amount` and `rate` in the `test` function still refer to their initial values!

```
def addInterest(balance, rate):
    newBalance = balance * (1 +
rate)
    balance = newBalance

def test():
    amount = 1000
    rate = 0.05
    addInterest(amount, rate)
    print amount
```

Python Programming, 1/e

60

Functions that Modify Parameters

- To summarize: the formal parameters of a function only receive the *values* of the actual parameters. The function does not have access to the variable that holds the actual parameter.
- Python is said to pass all parameters *by value*.

Python Programming, 1/e

61

Functions that Modify Parameters

- Some programming languages (C++, Ada, and many more) do allow variables themselves to be sent as parameters to a function. This mechanism is said to pass parameters *by reference*.
- When a new value is assigned to the formal parameter, the value of the variable in the calling program actually changes.

Python Programming, 1/e

62

Functions that Modify Parameters

- Since Python doesn't have this capability, one alternative would be to change the `addInterest` function so that it returns the `newBalance`.

Python Programming, 1/e

63

Functions that Modify Parameters

```
def addInterest(balance, rate):  
    newBalance = balance * (1 + rate)  
    return newBalance  
  
def test():  
    amount = 1000  
    rate = 0.05  
    amount = addInterest(amount, rate)  
    print amount  
  
test()
```

Python Programming, 1/e

64

Functions that Modify Parameters

- Instead of looking at a single account, say we are writing a program for a bank that deals with many accounts. We could store the account balances in a list, then add the accrued interest to each of the balances in the list.
- We could update the first balance in the list with code like:
`balances[0] = balances[0] * (1 + rate)`

Python Programming, 1/e

65

Functions that Modify Parameters

- This code says, “multiply the value in the 0th position of the list by (1 + rate) and store the result back into the 0th position of the list.”
- A more general way to do this would be with a loop that goes through positions 0, 1, ..., length - 1.

Python Programming, 1/e

66

Functions that Modify Parameters

```
# addinterest3.py
# Illustrates modification of a mutable parameter (a list).

def addInterest(balances, rate):
    for i in range(len(balances)):
        balances[i] = balances[i] * (1+rate)

def test():
    amounts = [1000, 2200, 800, 360]
    rate = 0.05
    addInterest(amounts, 0.05)
    print amounts

test()
```

Python Programming, 1/e

67

Functions that Modify Parameters

- Remember, our original code had these values:
[1000, 2200, 800, 360]
- The program returns:
[1050.0, 2310.0, 840.0, 378.0]
- What happened? Python passes parameters by value, but it looks like `amounts` has been changed!

Python Programming, 1/e

68

Functions that Modify Parameters

- The first two lines of `test` create the variables `amounts` and `rate`.
- The value of the variable `amounts` is a list object that contains four int values.

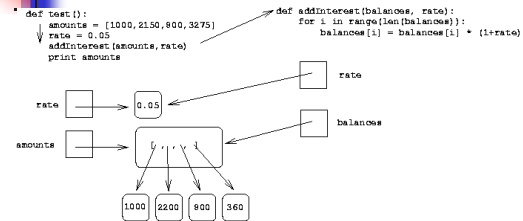
```
def addInterest(balances, rate):
    for i in range(len(balances)):
        balances[i] = balances[i] * (1+rate)

def test():
    amounts = [1000, 2200, 800, 360]
    rate = 0.05
    addInterest(amounts, 0.05)
    print amounts
```

Python Programming, 1/e

69

Functions that Modify Parameters



Python Programming, 1/e

70

Functions that Modify Parameters

- Next, `addInterest` executes. The loop goes through each index in the range 0, 1, ..., length -1 and updates that value in `balances`.

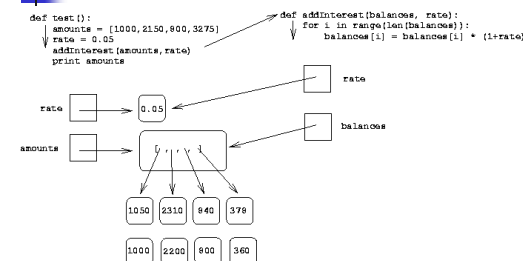
```
def addInterest(balances, rate):
    for i in range(len(balances)):
        balances[i] = balances[i] * (1+rate)

def test():
    amounts = [1000, 2200, 800, 360]
    rate = 0.05
    addInterest(amounts, 0.05)
    print amounts
```

Python Programming, 1/e

71

Functions that Modify Parameters



Python Programming, 1/e

72

Functions that Modify Parameters

- In the diagram the old values are left hanging around to emphasize that the numbers in the boxes have not changed, but the new values were created and assigned into the list.
- The old values will be destroyed during garbage collection.

```
def addInterest(balances, rate):
    for i in range(len(balances)):
        balances[i] = balances[i]
            * (1+rate)
def test():
    amounts = [1000, 2200, 800,
360]
    rate = 0.05
    addInterest(amounts, 0.05)
    print amounts
```

Python Programming, 1/e

73

Functions that Modify Parameters

- When `addInterest` terminates, the list stored in `amounts` now contains the new values.
- The variable `amounts` wasn't changed (it's still a list), but the state of that list has changed, and this change is visible to the calling program.

Python Programming, 1/e

74

Functions that Modify Parameters

- Parameters are always passed by value. However, if the value of the variable is a mutable object (like a list of graphics object), then changes to the state of the object *will* be visible to the calling program.
- This situation is another example of the aliasing issue discussed in Chapter 5!

Python Programming, 1/e

75

Functions and Program Structure

- So far, functions have been used as a mechanism for reducing code duplication.
- Another reason to use functions is to make your programs more *modular*.
- As the algorithms you design get increasingly complex, it gets more and more difficult to make sense out of the programs.

Python Programming, 1/e

76

Functions and Program Structure

- One way to deal with this complexity is to break an algorithm down into smaller subprograms, each of which makes sense on its own.
- This topic will be discussed in more detail in Chapter 9.

Python Programming, 1/e

77

Functions and Program Structure

```
def main():
    # Introduction
    print "This program plots the growth of a
10 year investment."

    # Get principal and interest rate
    principal = input("Enter the initial
principal: ")
    apr = input("Enter the annualized interest
rate: ")

    # Create a graphics window with labels on
left edge
    win = GraphWin("Investment Growth Chart",
300, 240)
    win.setBackground("white")
    win.setCoords(-1.75,-200, 11.5, 10400)
    Text(Point(-1, 0), " 0.0K").draw(win)
    Text(Point(-1, 2500), " 2.5K").draw(win)

    Text(Point(-1, 5000), " 5.0K").draw(win)
    Text(Point(-1, 7500), " 7.5K").draw(win)
    Text(Point(-1, 10000), "10.0K").draw(win)

    # Draw bar for initial principal
    drawBar(win, 0, principal)

    # Draw a bar for each subsequent year
for year in range(1, 11):
    principal = principal * (1 + apr)
    drawBar(win, year, principal)

    raw_input("Press <Enter> to quit.")
    win.close()
```

Python Programming, 1/e

78

Functions and Program Structure

- We can make this program more readable by moving the middle eight lines that create the window where the chart will be drawn into a value returning function.

Python Programming, 1/e

79

Functions and Program Structure

```
def createLabeledWindow():
    window = GraphWin("Investment Growth
    Chart", 320, 240)
    window.setBackground("white")
    window.setCoords(-1.75,-200, 11.5, 10400)
    Text(Point(-1, 0), ' 0.0K').draw(window)
    Text(Point(-1, 2500), ' 2.5K').draw(window)
    Text(Point(-1, 5000), ' 5.0K').draw(window)
    Text(Point(-1, 7500), ' 7.5K').draw(window)
    Text(Point(-1, 10000),
    '10.0K').draw(window)
    return window

def main():
    print "This program plots the growth of a
    10 year investment."
    principal = input("Enter the initial
    principal: ")
    apr = input("Enter the annualized interest
    rate: ")
    win = createLabeledWindow()
    drawBar(win, 0, principal)
    for year in range(1, 11):
        principal = principal * (1 + apr)
        drawBar(win, year, principal)
    raw_input("Press <Enter> to quit.")
    win.close()
```

Python Programming, 1/e

80