Functions That Return Values

- We’ve already seen numerous examples of functions that return values to the caller.
  
  \[ \text{discRt} = \text{math.sqrt}(b^2 - 4a^c) \]

- The value \( b^2 - 4a^c \) is the actual parameter of \texttt{math.sqrt}.

- We say \texttt{sqrt} returns the square root of its argument.
Functions That Return Values

- This function returns the square of a number:

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

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

```python
def sumDiff(x, y):
    sum = x + y
    diff = x - y
    return sum, diff
```
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).
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`!
Functions That Return Values

- If your value-returning functions produce strange messages, check to make sure you remembered to include the `return`!
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.
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
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:
  ```python
def test():
    amount = 1000
    rate = 0.05
    addInterest(amount, rate)
    print(amount)
```
Functions that Modify Parameters

- We hope that that the 5% will be added to the amount, returning 1050.

- ```
>>> test()
1000
``` 

- What went wrong? Nothing!
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.

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

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

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

def test():
    amount = 1000
    rate = 0.05
    addInterest(amount, rate)
    print(amount)
```
Functions that Modify Parameters

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

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

def test():
    amount = 1000
    rate = 0.05
    addInterest(amount, rate)
    print(amount)
```
Functions that Modify Parameters

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

def add_interest(balance, rate):
    new_balance = balance * (1 + rate)
    balance = new_balance
```

```plaintext
amount ➔ 1000
rate ➔ 0.05
balance ➔
```
Functions that Modify Parameters

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

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

def test():
    amount = 1000
    rate = 0.05
    addInterest(amount, rate)
    print(amount)
```
Functions that Modify Parameters

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

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

def test():
    amount = 1000
    rate = 0.05
    addInterest(amount, rate)
    print(amount)
```
Functions that Modify Parameters

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

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

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

def test():
    amount = 1000
    rate = 0.05
    addInterest(amount, rate)
    print(amount)```
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.
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.
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`. 
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()
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:
  ```python
  balances[0] = balances[0] * (1 + rate)
  ```
Functions that Modify Parameters

- This code says, “multiply the value in the 0\textsuperscript{th} position of the list by (1 + rate) and store the result back into the 0\textsuperscript{th} position of the list.”

- A more general way to do this would be with a loop that goes through positions 0, 1, \ldots, \text{length \textendash 1}.
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()
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!
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.

```python
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
```
Functions that Modify Parameters

```python
def test():
    amounts = [1000, 2150, 800, 3275]
    rate = 0.05
    addInterest(amounts, rate)
    print(amounts)

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

```
rate

0.05

amounts

[ ]

[1000, 2200, 800, 360]

balances

rate
```
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`.

```python
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
```
Functions that Modify Parameters

```python
def test():
    amounts = [1000, 2150, 900, 3275]
    rate = 0.05
    addInterest(amounts, rate)
    print(amounts)

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

Diagram:
- `def test()`: 
  - `amounts = [1000, 2150, 900, 3275]`
  - `rate = 0.05`
  - `addInterest(amounts, rate)`
  - `print(amounts)`

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

- `rate`
- `0.05`
- `amounts`
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.

```python
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
```
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.
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.