Chapter 6
Defining Functions
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.
Objectives (cont.)

- To write programs that use functions to reduce code duplication and increase program modularity.
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())
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.
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.
Functions, Informally

- **Happy Birthday lyrics…**
  ```python
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…
  ```

  ```python
>>> main()
Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Fred...
Happy birthday to you!
  ```
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:
  ```python
def happy():
    print "Happy birthday to you!"
  ```
- With this function, we can rewrite our program.
Functions, Informally

- The new program –
  ```python
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!
  ```
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()
Functions, Informally

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

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

- This gives us this new output

```python
>>> 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!
```
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.
Functions, Informally

- The generic function *sing*
  ```python
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.
Functions, Informally

- Our new output –
  ```python
  >>> 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!
Functions, Informally

- **Our new main program:**
  ```python
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!
  ```
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.
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
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.
Functions and Parameters: The Details

- A function is called by using its name followed by a list of actual parameters or arguments.
  \[
  \text{<name>)(<actual-parameters>)}
  \]

- When Python comes to a function call, it initiates a four-step process.
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.
Functions and Parameters: The Details

- Let’s trace through the following code:
  ```python
  sing("Fred")
  print
  sing("Lucy")
  ```
- When Python gets to `sing("Fred")`, execution of `main` is temporarily suspended.
- Python looks up the definition of `sing` and sees that it has one formal parameter, `person`. 
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:

  ```python
  person = "Fred"
  ```
Note that the variable `person` has just been initialized.
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`. 
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.
Functions and Parameters: The Details

- 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.
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”.
Functions and Parameters: The Details

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

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

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

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