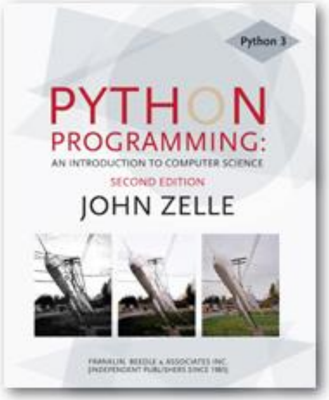
Python Programming: An Introduction to Computer Science


Python 3
PYTHON
PROGRAMMING:
AN INTRODUCTION TO COMPUTER SCIENCE
SECOND EDITION
JOHN ZELLE
FRANKLIN, BEEDLE & ASSOCIATES INC.
(INDEPENDENT PUBLISHERS SINCE 1985)

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())

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.

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.

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:

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

- With this function, we can rewrite our program.

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!
```

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()
```

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!
```

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`

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

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

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!

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!
```

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*.
  <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:

```
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:

```
person = "Fred"
```

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.

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.

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.

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
  ```

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
  ```

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
  ```

Functions That Return Values

- When calling this function, use simultaneous assignment.

```
num1, num2 = eval(input("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 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

- 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)
```

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.

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.

```
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

- 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)
```

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
```

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.

Boolean practice problem

- Write a program that reads a number from the user and tells whether it is prime, and if not, gives the next prime after it.
  - Example logs of execution: (run #1)

    ```
    Type a number: 29
    29 is prime
    ```

    (run #2)

    ```
    Type two numbers: 14
    14 is not prime; the next prime after 14 is 17
    ```

- As part of your solution, write two methods:
  - `isPrime`: Returns `true` if the parameter passed is a prime number
  - `nextPrime`: Returns the next prime number whose value is greater than or equal to the parameter passed. (If the parameter passed is prime, returns that number.)