Eric Roberts
CSCI 121

Handout #7
September 7, 2018

Functions

Functions

Eric Roberts
CSCI 121
September 7, 2018

Holism *vs.* Reductionism

In his Pulitzer-prizewinning book, computer scientist Douglas Hofstadter identifies two concepts—***holism*** and ***reductionism***—that turn out to be important as you begin to learn about programming. Hofstadter explains these concepts using a dialogue in the style of Lewis Carroll:

Achilles:	I will be glad to indulge both of you, if you will first oblige me, by telling me the meaning of these strange expressions, "holism" and "reductionism".
Crab:	Holism is the most natural thing in the world to grasp. It's simply the belief that "the whole is greater than the sum of its parts". No one in his right mind could reject holism.
Anteater:	Reductionism is the most natural thing in the world to grasp. It's simply the belief that "a whole can be understood completely if you understand its parts, and the nature of their 'sum'". No one in her left brain could reject reductionism.

A Quick Review of Functions

- You have been working with functions ever since Chapter 1.
- At the most basic level, a ***function*** is a sequence of statements that has been collected together and given a name. The name makes it possible to execute the statements much more easily; instead of copying out the entire list of statements, you can just provide the function name.
- The following terms are useful when working with functions:
  - Invoking a function by name is known as ***calling*** that function.
  - The caller passes information to a function using ***arguments***.
  - When a function completes its operation, it ***returns*** to its caller.
  - A function gives information to the caller by ***returning a result***.

Review: Syntax of Functions

- The general form of a function definition in Python is

```
def name(parameter list):
    statements in the function body
```

  where *name* is the name of the function, and *parameter list* is a list of variables used to hold the values of each argument.
- You can return a value from a function by including one or more `return` statements, which are usually written as

```
return expression
```

  where *expression* is an expression that specifies the value you want to return.

Predicate Functions

- Functions in Python can return values of any type. Those that return Boolean values play a central role in programming and are called ***predicate functions***. As an example, the following function tests if the first argument is divisible by the second:

```
def isDivisibleBy(x, y):
    return x % y == 0
```

- Once you have defined a predicate function, you can use it any conditional expression. For example, you can print the integers between 1 and 99 that are divisible by 7 as follows:

```
for i in range(1, 100):
    if isDivisibleBy(i, 7):
        print(i)
```

Using Predicate Functions Effectively

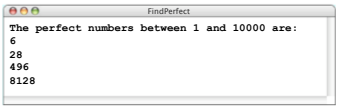
- New programmers often seem uncomfortable with Boolean values and end up writing really ugly code. For example, a beginner might write `isDivisibleBy` like this:

```
def isDivisibleBy(x, y):
    if x % y == 0:
        return True
    else:
        return False
```

  While this code is not technically incorrect, it is inelegant enough to deserve the bug symbol.
- A similar problem occurs when novices explicitly check to see whether a predicate function returns `True`. You should be careful to avoid such redundant tests in your own programs.

Exercise: Finding Perfect Numbers

- Greek mathematicians took a special interest in numbers that are equal to the sum of their proper divisors (a proper divisor of *n* is any divisor less than *n* itself). They called such numbers ***perfect numbers***. For example, 6 is a perfect number because it is the sum of 1, 2, and 3, which are the integers less than 6 that divide evenly into 6. Similarly, 28 is a perfect number because it is the sum of 1, 2, 4, 7, and 14.
- For the next several minutes of class, we're going to design and implement a Python program that finds all the perfect numbers between two limits. For example, if the limits are 1 and 10000, the output should look like this:



Functions that Return Graphical Objects

- Since return values can be of any type, functions can return graphical objects. For example, Figure 3-14 defines the following function, which returns a `GOval`:

```
def createFilledCircle(x, y, r, color):
    circle = GOval(x - r, y - r, 2 * r, 2 * r)
    circle.setColor(color)
    circle.setFilled(True)
    return circle
```

- Calling this function creates a circular `GOval` object of radius `r`, centered at `(x, y)` and filled with the specified color.
- You can use `createFilledCircle` to create as many circles as you need. You can create and display a filled circle in a single line, instead of the four lines you need without it.

The Purpose of Parameters

"All right, Mr. Wiseguy," she said, "you're so clever, you tell us what color it should be."
—Douglas Adams, *The Restaurant at the End of the Universe, 1980*

- As a general rule, functions perform a service for their callers. In order to do so, the function needs to know any details that are necessary to carry out the requested task.
- Imagine that you were working as an low-level animator at Disney Studies in the days before computerized animation and that one of the senior designers asked you to draw a filled circle. What would you need to know?
- At a minimum, you would need to know where the circle should be placed in the frame, how big to make it, and what color it should be. Those values are precisely the information conveyed in the parameters.

Keyword Arguments

- Python allows you to use either of two strategies to specify the arguments to a function:
  - Arguments may be specified by ***position***, in which case the parameter values are initialized in the order in which they appear: the first argument is copied into the first parameter variable, and so on.
  - Arguments may be specified by ***keyword***, in which the caller precedes the argument with a parameter name and an equal sign. In this case, the argument value is stored in the specified parameter.
- Positional arguments must precede keyword arguments.
- Keyword arguments may appear in any order.

Default Parameters

- Python allows you to specify a default value for a parameter by adding an equal sign and a value after the parameter name.
- For example, the following extension to `createFilledCircle` allows the caller to leave out the color of the fill, which defaults to `"Black"`:

```
def createFilledCircle(x, y, r, fill="Black"):
    circle = GOval(x - r, y - r, 2 * r, 2 * r)
    circle.setFilled(True)
    circle.setColor(fill)
    return circle
```

- A detailed explanation of how Python interprets positional, keyword, and default parameters appears on the next slide.

Mechanics of the Function-Calling Process

1. Evaluate the arguments in the context of the caller.
2. Reserve space for the function in a new ***stack frame***.
3. Copy each positional argument into the corresponding parameter variable.
4. Copy each keyword argument to the parameter with that name.
5. For parameters that include default values, assign those values to any arguments that are still unspecified. If any values are still unassigned after this step, Python reports an error.
6. Evaluate the statements in the function body, using the new stack frame to look up the values of local variables.
7. On encountering a `return` statement, compute the return value and substitute that value in place of the call.
8. Remove the stack frame for the called function.
9. Return to the caller, continuing from where the computation left off.

Extending `createFilledCircle`

- The next two lines walk through the steps involved in calling an extended version of the `createFilledCircle` function from Chapter 3. The new version has the following features:
  - Callers can leave out the fill color, in which case it defaults to "Black" (as shown on an earlier slide).
  - The fill and border colors may be specified independently.
  - The caller can use keyword arguments to specify the border and fill colors in either order.
- The code also illustrates the use of the `is` operator to test whether the value of the border parameter is `None`.
- Your task in these slides is to anticipate what happens next at each point in the function calling process.

Passing Parameters with Defaults

```
xcenter = gw.getWidth() / 2
ycenter = gw.getHeight() / 2
radius = 3 / 8 * gw.getHeight()
gw.add(createFilledCircle(xcenter, ycenter, radius, "Red"))
```

xcenter	ycenter	radius
250	100	75

```
def createFilledCircle(x, y, r, fill="Black", border=None):
    circle = GOval(x - r, y - r, 2 * r, 2 * r)
    circle.setFilled(True)
    if border is None:
        circle.setColor(fill)
    else:
        circle.setColor(border)
        circle.setFillColor(fill)
    return circle
```

x	y	r	fill	border

Passing Keyword Parameters

```
gw.add(createFilledCircle(250, 100, 75, border="Red", fill="Blue"))
```

```
def createFilledCircle(x, y, r, fill="Black", border=None):
    circle = GOval(x - r, y - r, 2 * r, 2 * r)
    circle.setFilled(True)
    if border is None:
        circle.setColor(fill)
    else:
        circle.setColor(border)
        circle.setFillColor(fill)
    return circle
```

x	y	r	fill	border

The Combinations Function

- To illustrate method calls, the text uses a function $C(n,k)$ that computes the ***combinations*** function, which is the number of ways one can select k elements from a set of n objects.
- Suppose, for example, that you have a set of five coins: a penny, a nickel, a dime, a quarter, and a dollar:

How many ways are there to select two coins?

penny + nickel	nickel + dime	dime + quarter	quarter + dollar
penny + dime	nickel + quarter	dime + dollar	
penny + quarter	nickel + dollar		
penny + dollar			

for a total of 10 ways.

Combinations and Factorials

- Fortunately, mathematics provides an easier way to compute the combinations function than by counting all the ways. The value of the combinations function is given by the formula

$$C(n,k) = \frac{n!}{k! \times (n-k)!}$$

- Given that you already have a `fact` function, is easy to turn this formula directly into a function, as follows:

```
def combinations(n, k):
    return fact(n) // (fact(k) * fact(n - k))
```

- The next slide simulates the operation of `combinations` and `fact` in the context of a call to `combinations(6, 2)`.

Tracing the Combinations Function

```
combinations(6, 2)
```

```
def combinations(n, k):
    return fact(n) // ( fact(k) * fact(n - k) )
}
```

n	k

```
def fact(n):
    result = 1
    for i in range(1, n + 1):
        result *= n
    return result
```

n	result	i