More on Functions

Feb 20th, 2012

Keyword arguments

- You can avoid matching by position by using *keyword arguments* in the function call.
- **Example:** `manyRandomWalks(numRepititions = 200, n = 20)`
- Here `numRepititions` and `n` are function parameters.
- Since the actual parameters are explicitly being provided values in the function call, the matching of arguments to parameters is no longer positional.
- The above function call is identical to the call `manyRandomWalks(n = 20, numRepititions = 200)`

Keyword parameters

- There is a way to define *default* values of parameters.
- **Example**: `def manyRandomWalks(n, numRepititions = 100)`
- This function can now be called with one or two arguments and in different styles.
- **Examples**: Try these out
  - `manyRandomWalks(10)`

    (The default value of 100 us used for `numRepititions`; 10 is used for `n`)
  - `manyRandomWalks(40, 150)`

    (40 is used for `n`, 150 for `numRepititions`)

Another example

```
def test(x = 3, y = 100, z = 200):
    return x - y + z
```

Examples of function calls:

1. test(10) (10 is used for x; default values 100 for y and 200 for z)
2. test(10, 20) (10 is used for x, 20 for y; default value 200 for z)
3. test(z = 35) (default values 3 for x, 100 for y; 35 for z)
4. test(10, z = 35) (10 for x, default value 100 for y, 35 for z)
5. test(z = 50, 10, 12) (Error: positional arguments come first, then keyword arguments)

Things that functions return

- Functions don't have to explicitly return values. For example:

```
def printGreeting(name):
        print "Hello", name, "how are you?"
```

- How would you call such a function?

Example:

```
printGreeting("Michelle")
```

- What would happen if you executed?

```
x = printGreeting("Michelle")
```

The object None

- It is used by Python to represent the absence of a value.
- It has a type called NoneType and None is the only object of this type.
- None has a boolean value that is False.

Functions practice problem 1

- Write a function called `search` that reads a sequence of words (strings) one per line, looking for the word "hello." The function should assume that the sequence will be terminated by the empty string.

- **Enhancements**:

1. Make the function have a keyword parameter that represents the word it is searching for. Have the default value of this be "hello."
2. Make the function have an additional parameter that represents the number of words it is willing to read while waiting for the word it is looking for.

Functions Practice Problem 2

- Write a function that simulates the roll of two 6-sided dice 100 times and returns the number of times 4 shows up as the sum of the outcomes on the two dice.
- **Enhancements**:

1. Make the function take the number of times it needs to roll the dice as a parameter, with 100 being the default value.
2. Make the function take the number of sides of the die as a parameter, with 6 being the default value.
3. Make the function take the number of dice it needs to roll as a parameter, with 2 being the default value.

Ordering functions in your code

- Will the following code work? Here the function is defined after the main program that is calling it.

```
print foo()
def foo():
        return "hello"
```

- Will this work? Here functions are defined before the main program. But, `foo2()` is called before it is defined by `foo1`.

```
def foo1():
        return foo2()
def foo2():
        return "hello"
print foo1()
```

How does Python process code with functions?

```
def foo1():
        return foo2()
def foo2():
        return "hello"
print foo1()
```

1. Python starts scanning the code from the beginning of the file.
2. It notes down names of functions as it encounters their *definitions*. Note that the functions are not executed at this time.
3. It reaches the first executable statement (`print foo1()`) and since `foo1` is known to Python, control is transferred to `foo1`.
4. In `foo1`, Python encounters a call to `foo2`. Function `foo2` is also known to Python and so control is transferred to `foo2`.

Moral of this example?

- Define *all* functions before the main program.

- And then don't worry about the order in which the functions themselves are defined.

Scope of a variable

- The *scope* of a variable refers to the “where” and “when” a variable is available for use.
- Things were simple when we did not have functions.
- If we only had a main program: the scope of a variable extends from the point where the variable is first defined till the end of the program.
- In Python the scope of a variable can be *dynamic*.

Example of dynamic scope

```
x = raw_input()
if x:
    y = "hello"
print y
```

- If the input is a non-empty string, then the scope of variable y starts at Line 3. Otherwise, the scope of y is empty, i.e., y is undefined.

Scope of variables inside functions

- Parameters and variables defined inside a function are “local” to that function.

```
def foo():
    var1 = “hello”
    return var1 + var1

# main program
print foo()
if var1 == “hellohello”:
    print foo()
```

var1 is a variable that is local to foo(). It comes into existence when the first line of foo() is executed and it “dies” when we exit the function.

var1 is not defined and this usage will cause an error.

Function parameters are also local

```
def foo(x):
    var1 = “hello”
    return var1 + x

# main program
print foo(“bye”)
if x == “hellohello”:
    print foo()
```

The variable x is undefined here because the parameter x lives only for the duration of the function

Mental model: version 1

1. Python creates a dictionary of variable names when it starts evaluating the main program. It uses this dictionary to insert, look up, and update variable names.
2. When the function `foo` is executed, a new dictionary of variable names, specific to `foo` is created.
3. First the parameter `x` is inserted into this dictionary. Then variable `var1` is inserted.
4. Whenever we access a variable inside `foo`, `foo`’s dictionary is looked up.
5. When the execution of `foo` is over, `foo`’s dictionary is destroyed.