More about functions

FEB 20TH, 2013

The manyRandomWalks functions

- Definition:

```
def manyRandomWalks(n, numRepititions):
        ...
        ...
        return float(sum)/numRepititions
```

- The first line of the function definition is called the *function header*. The rest of the function is called the *function body*.
- The names `n` and `numRepititions` in the function header are called *parameters* of the function.
- In a the following call to this function:

```
print manyRandomWalks(m, 100)
```

- The expressions `m` and `100` are called function *arguments*.

Parameters versus Arguments

- Parameters are variables used in a function header.
- Parameters get assigned values when a function is called.

```
def foo(x, y, z):
    x = y + z
    return x + y + z
```

- Here `x`, `y`, and `z` are parameters of the function `foo`.
- Inside the function `foo`, they can be treated as variables that acquire values provided by a function call (e.g., `foo(2, 7, 3)`).

Parameters versus Arguments

- Arguments in a function call could be complicated expressions that will be evaluated to a value first before being sent in to the function.

  Example: `manyRandomWalks(80/x, y + 1)`

- In fact, arguments could be expressions involving calls to other functions.

  Example: `manyRandomWalks(int(math.sqrt(x)), y + 1)`

Matching arguments to parameters

- One way in which Python matches arguments to parameters is by reading them left to right and matching 1st argument to 1st parameter, 2nd argument to 2nd parameter, etc.

- This is called the *positional style* of parameter passing.

- So

  ```
  manyRandomWalks(10, 100)
  ```

  and

  ```
  manyRandomWalks(100, 10)
  ```

  will return very different values.

- In this way of parameter passing the number of arguments and the number of parameters also have to exactly match.

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

- None is a built-in constant in Python that is used to indicate the absence of a value.

- In the example,

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

  x is assigned the value None. You can see this by trying

  ```
  print x
  ```

- To understand None better try:
  - type(x)
  - bool(x)

- Unline True and False which can be assigned to even though they are listed as built-in Python constants, None cannot be assigned to.

Solution to Quiz 4 Problem

- Define a function called `factorSum` whose header is:

  `def factorSum(n):`

- The parameter `n` is expected to be a positive integer and the function returns the sum of all the factors of `n`. For example, if `n` were 10, the function would return 18 (which is 1 + 2 + 5+ 10).

(In the quiz, you were asked to assume that such a function was already provided to you.)

Function factorSum

```
# Programmer: Sriram Pemmaraju
# Date: Feb 19th, 2013

# This function takes a postive integer parameter and
# returns the sum of the factors (1 and n included) of
# n.
def factorSum(n):
    factor = 1 # tracks potential factors from 1 through n
    sumOfFactors = 0 # tracks the sum of factors
    while factor <= n:
        # I use a slightly obscure style here to illustrate the fact
        # that all objects in Python have boolean values.
        # Here, if factor evenly divides n, then n%factor evaluates to
        # 0, which has boolean value False and therefore in this case
        # not n%Factor evaluates to True
        if not n%factor:
            sumOfFactors = sumOfFactors + factor
        factor = factor + 1

    return sumOfFactors
```

Problem (continued)

- A positive integer n is called perfect if the sum of all the factors of n, excluding n, is equal to n. For example, n = 6 is perfect because its factors, excluding 6, are 1, 2, and 3 and 1+ 2 + 3 = 6. For this problem, we want you to write a function called nextPerfect with the following function header:

```
def nextPerfect(M):
```

- You can assume that the parameter M is a positive integer. This function should return the smallest perfect integer that is greater than or equal to M. This function should call factorSum repeatedly to complete its task.

- **Example.** The first four perfect integers are 6, 28, 496, and 8128. So if we call nextPerfect(10), it should return 28. Even if we call nextPerfect(20), the function should return 28. In fact, even if we call nextPerfect(28), the function should return 28. However, if we call nextPerfect(29), the function should return 496.

Function nextPerfect

```
# This function returns the smallest perfect number
# that is greater than or equal to M.
def nextPerfect(M):

    while True:
        # factorSum(n) returns the sum of all factors of n, including
        # n. So n has to be first excluded from this sum before we compare
        # it to n.
        if factorSum(M) - M == M:
            return M
        M = M + 1
```