CS 115 Lecture 12

Functions

Neil Moore

Department of Computer Science
University of Kentucky
Lexington, Kentucky 40506
neil@cs.uky.edu

22 Oct 2015
27 Oct 2015

Factoring ifs

Before we start with functions...
Sometimes you may find yourself repeating the same code in both branches of an **if..else**.

- Repeated code is bad!
- But it has to happen in both cases, so what to do?
- "Factor it out"
  - If the repeated code is at the beginning, move it before the if.
  - If it is at the end, move it to after the else.
  - If it is in the middle, you can split the if/else in half.
    - So you will have two if/else statements.
    - Put the repeated code between .

```
if user_is_american:
    input = feet2meters(input)
    some long calculation
    print(meters2feet(result))
else:
    some long calculation
    print(result)
```

With the duplication factored out:

```
if user_is_american:
    input = feet2meters(input)

some long calculation

if user_is_american:
    print(meters2feet(result))
else:
    print(result)
```

The idea behind functions

Sometimes we need to repeat the same combination of control structures and steps in several different places.

- **Functions** let you write the code once...
- ...then use it in multiple places.
- It's like defining a new verb.
- We've already seen several built-in and library functions:
  - `print`, `input`, `range`
  - `math.sin`, `random.randrange`
  - Constructors: `GraphWin`, `Rectangle`
  - Methods: `setText`, `getMouse`
- And we know how to call them:
  - If it doesn't return a value: `func(`*arguments*`)`
  - If it does return a value: `result = func(`*arguments*`)`
- We've even written one of our own!
  - `def main():` defines a function named **main**.

Calling a function: syntax

To call ("invoke") a function, you write its name, followed by its **arguments** in parentheses.

- The name of the function is just an identifier.
  - Same as variables.
  - Letters, digits, and underscores; can't start with a digit.
- The arguments are a list of expressions, separated by commas.
  - Arguments are "inputs" that the program sends to the function.
  - Each function specifies how many arguments it takes, of what types.
  - Some, like `random.random`, take no arguments.
    - Still need parentheses!

Calling a function: semantics

When you call a function:

- The interpreter pauses executing the code that had the function call.
  - But it remembers where it was.
- The interpreter runs the code of the called function.
  - It makes the arguments available to the function.
    - We'll see how that works soon.
- The function finishes or **returns**.
  - The interpreter picks up where it left off.
  - The value of the function call is whatever the function returned.
- Structured programming guarantee: When a subprogram (function) finishes, it returns to where it was called from.

Defining a function

To define a function, you need to specify three things:

- The name of the function.
- What arguments the function takes.
  - With a name for each argument.
  - Called the **parameter list**.
- What the function does when called.
  - The **code** or **body** of the function.
  - An indented block.

Defining functions

```
def name(parameters):
    body
```

- Defines a function named “name”.
- Goes at the **top level** of the file
  - Unindented.
  - Not inside `main` or other functions!
- Parameters are a comma-separated list of identifiers.
  - The function takes one argument for each parameter.
- When Python sees the definition, it doesn’t run the body!
  - Instead, it remembers it and gives it a name.
  - The body runs when you *call* the function.
  - That’s why we needed `main()` at the end of our programs.
  - If you don’t call a function, it will not run!
    - Something to check for when debugging.

A simple function definition

```
def triangle():  # No parameters.
    print("*")
    print("**")
    print("***")
```

Now when you call `triangle()`, it prints those three lines.

A simple function definition

Function definitions can contain any control structure you like.

```
def triangle():
    for i in range(1, 3 + 1):
        print("*" * i)
```

This function does the same thing as the previous version.

- "Multiplying" a string by an integer repeats the string.
  - `'*' * 3` evaluates to the string `'***'`

Function arguments

Most functions don't do exactly the same thing every time.

- Instead, they take **arguments** that control the details of what happens.
  - `print`: What value to print.
  - `input`: What prompt to display.
  - `randrange`: What range to pick the random number from.
- We use arguments to send information into a function.
  - They are the function's "inputs".
  - (Not the same as user input!)

Parameters and arguments

In the function definition, we create placeholders for the arguments, called **parameters**.

- Inside the function, parameters work like variables.
- When you call the function, each argument's value is stored in the corresponding parameter.
  - It is an error if you give more or fewer arguments than parameters.
- **Call by value**: the parameter holds the argument's value, but isn't the same as the argument.
  - If you assign a value to the parameter, that doesn't change the argument!
  - We'll see later that the situation is is more complicated when you have mutable arguments like lists or graphics shapes.

A function with a parameter

Let's change our `triangle` function to take one argument, the size of the triangle (number of lines).

```
def triangle(size):
    for i in range(1, size + 1):
        print("*" * i)
```

Now `triangle(3)` will print the three-high triangle.

- By setting `size = 3` before executing the body.
  - Remember the terminology!
  - `size` is the parameter
  - 3 is the argument (the value we give the parameter this time)
- Note that `size` is *only* accessible inside the function!
  - If you want to "set" it from elsewhere, call the function!

Multiple parameters

Functions can have more than one parameter.

```
def triangle(size, letter):
    for i in range(1, size + 1):
        print(letter * i)
```

```
triangle(2, "+")
+
++
```

- When calling the function, you must supply the same number of arguments as parameters.
  ```
  triangle(3) # ERROR
  triangle(3, "+", "-") # ERROR
  ```
- If arguments are out of order, the function does the wrong thing!
  ```
  triangle("+", 3) # ERROR in size + 1
  ```
- It's possible to have optional parameters in Python, but we won't use them in CS 115.

Returning a value

A function can send a value back to the caller by **returning**.

- Syntax: `return` *`expression`*
  Or plain: `return`
- Semantics:
  - First evaluates the expression if any.
  - Then ends the function's execution immediately (!!)
  - The value of the function call is the value of the expression, or the special value `None` if there was none.
- The expression is the "result of" the function.
- Remember the structured programming guarantees.
  - Each control structure should have *one exit*.
  - So in structured programming, a function should have **one return**, at the very end.
  - If the function doesn't have a value, either plain `return` at the end, or no return at all.

Parameters/return values vs input/output

- **Parameters** take "input" from **the rest of the program**.
- **Return values** send "output" to **the rest of the program**.
- `input()` takes input from **the user**.
- `print()` sends output to **the user**.
- Good functions use parameters and return values, not `input` and `print`.
  - Then you can use them not only with user input...
  - ...but also with graphical programs, files, computed data...
- When should a function use `input` or `print`?
  - When the function's sole purpose is to interact with the user.

A function that returns a value

```
def triangular_number(num):
    sum = 0
    for i in range(1, num + 1):
        sum += i
    return sum
```

When we call the function, run it and see what it returns:

```
print(triangular_number(5))
  ...  return 15
```

The interpreter plugs that value into the expression:

```
print(15)
```

15

Local variables

All the variables defined in a function are **local** to that function.

- For example, `i` in our triangle functions.
- They only exist while the function is running.
  - **Lifetime** or **extent**: the time during which a variable takes up memory.
  - Variable is "born" when it is initialized...
  - ...and "dies" when the function it is in returns.
- Other functions cannot see local variables at all!
  - The **scope** of the variable is what part of the code can see it.
  - The body of the function it is in, starting from the initialization.
  - Scope doesn't care about what else the function calls!
- This means your functions cannot refer to variables in other functions.
- If you need to access information from multiple functions:
  - Use a parameter and argument to send information into a function.
  - Use the return value to get information back out.
- A **global variable** is defined outside any function.
  - Avoid these! Very hard to reason about: could change at any time!

Where to put function definitions

- Functions (at least in this class) should be defined at the **top level** of your source file.
  - Not indented.
  - So not inside of another function like `main`.
- Functions must be defined before the code that calls them executes.
  - But if that code is inside another function, it only executes when the other function is *called*.
- So if function `main` calls function `triangle`:
  - `def triangle` must come before the call `main()`.
  - But it's okay if it comes after `def main`.
- In general: put the call `main()` at the end and you'll be fine.

A complete program with functions

```
def main():
    number = int(input("Enter a number:"))
    limit = square(number)
    print("Counting up to", limit)
    countto(limit)

def square(x):
    return x ** 2

def countto(number):
    for i in range(1, number):
        print(i, end = ", ")
    print(number)

main()
```

Let's try rearranging the functions.

Documenting functions

Functions are "subprograms", so should be documented like programs.

- Write a **header comment** for each function.
  - Three parts: **purpose**, **preconditions**, **postconditions**.
  - You should still have the usual comments in the function's code.
- Purpose: Describes what the function is supposed to do, *in terms of the parameters*.
  - *Not* where it is used: `sqrt`'s documentation does not say "used in the calculation of distance".

```
# Purpose:  Compute the square of the number x.
# Purpose:  Print a triangle of stars with size rows.
```

- Preconditions: What has to be true before the function is called?
  - What should be the type of each parameter?
  - Are there any other restrictions on the parameter values?

```
# Preconditions:  x is an integer or float.
# Preconditions:  size is a positive integer.
```

Documenting functions

- Postconditions: What will be true after the function finishes?
  - What type will be returned, if any?
  - Any other promises we will make about the return value?
  - What user input/output will the function have done?

```
# Postconditions:  Returns a positive number
#     of the same type as x.
# Postconditions:  Prints size lines to standard output.
```

- Preconditions and postconditions are like a legal contract:
  - "If you give me this (pre), I promise to give you that (post)."
  - Helps to identify where a bug is. If a function does the wrong thing:
    - ★ If the preconditions are all satisfied, it's a bug in the function.
    - ★ If they are not, it's a bug in the *caller*.

Where to put function documentation

- In most languages, the purpose/preconditions/postconditions are in a comment just before the function.
- Python has another way to do it: documentation strings.
  - These go inside the function, as the very first thing.
    - So must be indented!
  - Start with ''' (three single quotes)
  - Continues across multiple lines until another '''
- Documentation strings are not really comments.
  - They still don't do anything, but the interpreter does see them.
  - The `help` function in the Python shell displays a function's documentation string.

    ```
    help(square) # Just the function name, not a call.
    ```

    - You must execute the program first so the function is defined!
- Example: `countsquare-doc.py`

Unit testing

Functions can also be tested in the same way as whole functions.

- **Unit testing**: Testing individual functions in isolation.
  - Verify that the function keeps its promises.
- Just like a test case for a program. Three columns in your test plan:
  - Description of the test case.
    - "Normal case, positive integer."
  - Inputs: All the arguments you will pass to the function.
    - The preconditions.
    - Also list user input you will provide.
    - Be sure to indicate which is which!
  - Expected output: What the function should return and output.
    - The postconditions.
    - Be sure to distinguish the return value from printed output.

Driver functions

Once the test plan was written, doing the whole-program tests was easy.

- Just run the program, type the input, verify the output.
- But how do we do that with functions?
- One way: call the function from the shell window.
  - The interpreter will print the return value as the last thing.
  - Must execute the program first to define the functions!
- Another way: write a **driver** function.
  - A function like `main`: named `test` for example.
  - But instead of following your design, it just calls the functions with the inputs in the unit test.
    - Can even check whether the function returned the right thing!
    - This is one place where it's okay to hard-code values.
  - When you want to run your unit tests, just change `main()` to `test()`
- An example: `countsquare-test.py`

Advantages of functions

- They avoid repeated code.
- Allow you to re-use the same code in a later program.
- They provide more structure to programs.
  - The details of complicated procedures are hidden away.
    - Can choose to look at the details or the big picture.
  - Each piece of code deals with one topic.
    - Makes it easier to focus on one part at a time.
    - And for different programmers to work on different parts of the same project.
- They are easier to test and debug.
  - You can focus on testing one piece at a time.
  - Then put those well-tested pieces together into larger units.
  - Testing at every step of development means less time debugging!

Disadvantages of functions

- Functions are slightly slower than embedding the code.
  - Arguments have to be copied into parameters, etc.
  - Some languages (especially compiled ones) can avoid this.
- Every function takes a little bit of memory while it is running.
  - Parameters, local variables, "where was I?"
  - If functions call functions hundreds of calls deep, it could add up.
  - Usually only comes up with recursion (a function calling itself).
- More lines of code if you only call it once.
  - Two or three extra lines: **def:**, return, and calling the function.
  - If you call it many times, then it usually saves lines!

Usually the benefits are much higher than the costs!

Let's come back to the Collatz conjecture

Suppose we want to check all the numbers in a range to see if they satisfy the Collatz conjecture.

- `collatz2.py`
- Note the changes:
  - Moved the $3n+1$ loop into a function.
    - And removed the output of all the steps.
    - We just want True or False.
  - Made `iseven` into a function.
  - Added a while loop in `main`.
  - Gave `input_int` a parameter for the prompt.

Debugging functions: the call stack

What happens if we set a breakpoint in `iseven`?

- How did we get there?
  - Called by `collatz`.
  - So we'll go back there when `iseven` returns.
- How did we get to `collatz`?
  - Called by `main`.
  - So we'll go back there when `collatz` returns.
- When functions return, the interpreter goes back to the caller.
  - So the interpreter has to keep track of all the callers.
  - Who called me, who called them, who called them, ...
- The interpreter tracks this information with a data structure called the **call stack**.
  - Each item on the call stack is a function call in progress.
  - Stores the local variables for that function.
  - Calling: put a new item on top of the stack.
  - Returning: remove the item from the top of the stack.
  - The debugger can show it to you.

Single-stepping with functions

Remember the three kinds of stepping in the debugger. They have to do with functions.

- Step into: Pause at the next line (maybe in another function).
- Step over: Pause at the next line of *this* function.
- Step out: Pause when this function returns.

Let's see those in action while watching the call stack.

Tracing functions

What if we wanted to trace a program with functions?

- Our traces have a column for line number.
  - We'll be jumping around, so let's write the function name too.
- How do we show local variables?
  - Could have columns for *all* of them...
  - ...but that could be a lot!
    - (and doesn't work if we have recursive functions)
  - Instead we'll give each function call its own section of the trace.
    - With its own local variables as columns.
  - We'll indicate when we call a function or return from one.
    - Each time we do, include the headers for this function's variables.

Tracing functions: example

Let's trace this small program:

```
def triangular(num):
    sum = 0
    for i in range(1, num + 1):
        sum += i
    return sum

def pyramidal(n):
    sum = 0
    for i in range(1, n + 1):
        sum += triangular(i)
    return sum

def main():
    pyr = pyramidal(2)
    print(pyr)

main()
```

Trace of the function

main	pyr	Action/Output
12	–	(call `pyramidal(2)`)

pyramidal(2)	n	sum	i	Action/Output
6	2	–	–	
7	2	0	–	
8	2	0	1	
9	2	0	1	(call `triangular(1)`)

triangular(1)	num	sum	i	Action/Output
1	1	–	–	
2	1	0	–	
3	1	0	1	
4	1	1	1	
5	1	1	1	(return 1)

pyramidal(2)	n	sum	i	Action/Output
9	2	1	1	
8	2	1	2	

Trace, continued

pyramidal(2)	n	sum	i	Action/Output
9	2	1	1	(call `triangular(2)`)

triangular(2)	num	sum	i	Action/Output
1	1	–	–	
2	1	0	–	
3	1	0	1	
4	1	1	1	
3	1	1	2	
4	1	3	2	
5	1	3	2	(return 3)

pyramidal(2)	n	sum	i	Action/Output
9	2	4	1	
10	2	4	1	(return 4)

main	pyr	Action/Output
12	4	
13	4	4