ECOR 1041
Computation and Programming

Defining Functions

Carleton University

References

- *Practical Programming*, 3rd ed.
  - Chapter 3, pp. 35 - 46 and pp. 60 – 62
    - This is the following sections of Chapter 3:
      - Defining Our Own Functions
      - Using Local Variables for Temporary Storage
      - Tracing Function Calls in the Memory Model
    - And:
      - Omitting a return Statement: None
      - Dealing with Situations That Your Code Doesn’t Handle

Lecture Objectives

- Learn how to define and interactively test Python functions
- Extend the memory model to enable us to trace and visualize the execution of code that contains functions

Learning Outcomes (Vocabulary)

- Know the meaning of these words and phrases
  - Function definition (header, parameter, body)
  - Keyword (`def`, `return`)
  - Execute a function definition
  - Function call, call expression
  - Function argument
  - Activation frame, local variable

Learning Outcomes

- Implement mathematical functions as Python functions
- Test the functions interactively, using the Python shell
- Trace short programs that contain functions “by hand”, explain what happens as the computer executes each statement, and draw diagrams that depict the variables and parameters in the activation frames and the objects that are bound to the variables

Defining Functions

- Here is a mathematical function, f, that maps the radius of a sphere, r, to the volume of the sphere:

  $$f : r \rightarrow f(r), \text{ where } f(r) = 4/3 \times \boldsymbol{\pi} r^3$$

- How do we turn this into a Python function that takes a radius and produces a volume?

Defining Functions

- Shell experiment: try defining the function using “math notation”

```
>>> import math
>>> f(r) = 4 / 3 * math.pi * r ** 3
Syntax Error: cannot assign to a function call
```

- We will now learn how to write a Python function definition, using the volume-of-sphere function as an example

Defining Functions: Header

```
import math

def f(r):
    return 4 / 3 * math.pi * r ** 3
```

The function *header* begins with the *keyword* `def`

Defining Functions: Header

```
import math

def f(r):
    return 4 / 3 * math.pi * r ** 3
```

`def` is followed by the function name, `f`, then the *parameter*, `r`, enclosed in parentheses

Defining Functions: Header

```
import math

def f(r):
    return 4 / 3 * math.pi * r ** 3
```

The header ends with a colon

Defining Functions: Body

```
import math

def f(r):
    return 4 / 3 * math.pi * r ** 3
```

The function *body* contains the expression that calculates the value produced by the function

Defining Functions: Body

```
import math

def f(r):
    return 4 / 3 * math.pi * r ** 3
```

`return` (another keyword) evaluates the expression, ends the function's execution, and makes the value available as the value of the function call

Defining Functions

- Syntax rule: a function’s body must be indented relative to the function header
- Convention: the function body is indented four spaces

Math vs. Python

- Compare the mathematical notation to the function definition

$$\underline{f} : \underline{r} \rightarrow f(r), \text{ where } f(r) = \underline{4/3 \times \boldsymbol{\pi} r^3}$$

```
def f(r):
    return 4 / 3 * math.pi * r ** 3
```

Use Coding Conventions

- Rewrite the function (use descriptive names) so that it is easier to understand

```
def volume_of_sphere(radius):
    return 4 / 3 * math.pi * radius ** 3
```

Defining Functions in the Shell

- We can type the function definition in the shell

```
>>> def volume_of_sphere(radius):
...     return 4 / 3 * math.pi * radius ** 3
```

- Then, call the function (type the call expression)

```
>>> volume_of_sphere(1)
4.1887902047863905
```

The value of the call expression (displayed in the shell) is the value the function calculates and returns

Defining Functions in the Shell

- Drawbacks:
  - Function definitions typed are lost every time the shell restarts
  - Editing function definitions (fixing mistakes) is tedious

Defining Functions in a File

- Type the definition in an editor, save the code in a .py file

```
def volume_of_sphere(radius):
    return 4 / 3 * math.pi * radius ** 3
```

- Load it into the Python interpreter, then use the shell to call the function

```
>>> volume_of_sphere(1)
4.1887902047863905
```

Testing Functions Interactively

- Step 1: Calculate (using pencil, paper and a calculator) the values we expect the function to produce when it is called with different arguments
- Step 2: Use the shell to call the function for each of the arguments from Step 1 and compare the actual values returned by the function to the expected results

Testing Functions Interactively

```
>>> volume_of_sphere(1)
4.1887902047863905
>>> volume_of_sphere(2)
33.510321638291124
>>> volume_of_sphere(0)
0.0
>>> volume_of_sphere(-1)
-4.1887902047863905
```

Oops! How do we know not to use a negative argument? Later…

Python Tutor: Tracing Function Execution

- We will use Python Tutor to help us understand what happens when Python executes the `volume_of_sphere` example
  - A link is posted on Brightspace
- Python Tutor does not have a shell, so we type a complete program (function definitions and function calls) in the editor window

Python Tutor: Tracing Function Execution

- What happens if we change the return to a print?
  - The Python Tutor link is posted on Brightspace
- Functions should have print statements only if their “job” is to print something
- Mathematical functions like this one should use return and **not** print!!!

Python Tutor: What We Learned

- First, Python executes the function definition, which creates a *function object*
  - This object is assigned to a variable with the same name as the function, in the global frame
  - Executing the function definition <u>does not</u> call the function

Python Tutor: What We Learned

- Python executes the function call
  - The arguments in the call expression are evaluated
  - A new *activation frame* is created
  - Parameters are created in the frame.
    - Argument values are “passed into” the function by assigning them to the parameters

Python Tutor: What We Learned

- Python executes the function body
  - The expression after `return` is evaluated
  - The reference to this value is “returned” to the caller; i.e., is used as the value of the call expression
  - The activation frame is removed
- Execution continues with the statement after the function call

Local Variables

- Functions can contain assignment statements that create new variables in a function’s activation frame
- These are known as *local variables*, because they can only be accessed by statements in the function body

Example from *Practical Programming*

```
>>> def quadratic(a, b, c, x):
...     first = a * x ** 2
...     second = b * x
...     third = c
...     return first + second + third
...
```

- `first`, `second` and `third` are local variables, as are parameters `a`, `b`, `c` and `x`

Example from Practical Programming

- Parameters `a`, `b`, `c` and `x` are created in the activation frame that is created when `quadratic` is called
- Variables `first`, `second` and `third` are created in the activation frame when the assignment statements are executed
- The parameters and local variables disappear when `quadratic` returns and its activation frame is removed

Example from Practical Programming

- The parameters and local variables cannot be accessed from outside the function, because they exist only while the function is being executed

```
>>> quadratic(2, 3, 4, 0.5)
6.0
>>> first
NameError: name 'first' is not defined
>>> a
NameError: name 'a' is not defined
```

When Should We Use Local Variables?

- *Practical Programming* states, “breaking [computations] into several steps can lead to clearer code”
- This is correct, but many steps, with many local variables, can result in code that is difficult to understand
- Rewrite quadratic to use 6 local variables (next slide)

When Should We Use Local Variables?

```
>>> def quadratic(a, b, c, x):
...     first = x ** 2
...     second = a * first
...     third = b * x
...     fourth = second + third
...     fifth = c
...     sixth = fourth + fifth
...     return sixth
...
```

- Several tiny steps, but is the function easier to understand?

When Should We Use Local Variables?

- Are the local variables in the book's `quadratic` function required?
- We can replace the function body with a single `return` statement

```
>>> def quadratic(a, b, c, x):
...     return a * x ** 2 + b * x + c
...
```

No `return` Statement

- What happens if a function does not have a `return` statement?

```
def volume_of_sphere(radius):
    volume = 4 / 3 * math.pi * radius ** 3

>>> vol = volume_of_sphere(1)
>>> vol
>>>          # Nothing is displayed?
```

No `return` Statement

```
>>> vol = volume_of_sphere(1)
>>> print(vol)
None
```

- A function that does not have a `return` statement returns the value `None`
- When the value of an expression is `None`, it is not displayed by the shell

Recap of Learning Outcomes

Learning Outcomes (Vocabulary)

- Know the meaning of these words and phrases
  - Function definition (header, parameter, body)
  - Keyword (`def`, `return`)
  - Execute a function definition
  - Function call, call expression
  - Function argument
  - Activation frame, local variable

Learning Outcomes

- Implement mathematical functions as Python functions
- Test the functions interactively, using the Python shell
- Trace short programs that contain functions “by hand”, explain what happens as the computer executes each statement, and draw diagrams that depict the variables and parameters in the activation frames and the objects that are bound to the variables