

Python Programming: An Introduction to Computer Science

Chapter 6 Functions, Variables, Modules



Coming up: The Function of
Functions

1

The Function of Functions

- Why use functions at all?
 - Reduces duplicate code
(Less maintenance, debugging, etc...)
 - Makes programs easier to read
 - Makes programs more “modular”.. easier to change and reuse parts.

Coming up: Example Function versus No Functions

2

Example Function versus No Functions

See functionsexample.py

```
p1Name = raw_input("What is your name player1 ?")
p1Age = input("What is your age player1 ?")
p1Color = raw_input("What is your favorite color player1 ?")
p2Name = raw_input("What is your name player2 ?")
p2Age = input("What is your age player2 ?")
p2Color = raw_input("What is your favorite color player2 ?")
p3Name = raw_input("What is your name player3 ?")
p3Age = input("What is your age player3 ?")
p3Color = raw_input("What is your favorite color player3 ?")
print "Player 1 is %s who is %d years old. \nTheir favorite color is %s" \
      %(p1Name, p1Age, p1Color)
print "Player 2 is %s who is %d years old. \nTheir favorite color is %s" \
      %(p2Name, p2Age, p2Color)
print "Player 3 is %s who is %d years old. \nTheir favorite color is %s" \
      %(p3Name, p3Age, p3Color)
```

Coming up: Example Function versus No Functions

3

Example Function versus No Functions

```
# Get the player's information
def getInfo(playerNum):
    playerStr = str(playerNum)
    nm = raw_input("What is your name player"+playerStr+" ?")
    age = input("What is your age player"+playerStr+" ?")
    color = raw_input("What is your favorite color player"+playerStr+" ?")
    return nm, age, color

# Print out the information about a player
def printInfo(nm, age, color, num):
    print "Player %d is %s who is %d years old. \nTheir favorite color is %s" \
          %(num, nm, age, color)

def main():
    p1Name, p1Age, p1Color = getInfo(1)
    p2Name, p2Age, p2Color = getInfo(2)
    p3Name, p3Age, p3Color = getInfo(3)

    printInfo(p1Name, p1Age, p1Color, 1)
    printInfo(p2Name, p2Age, p2Color, 2)
    printInfo(p3Name, p3Age, p3Color, 3)

main()
```

Coming up: Types of Functions

4

Types of Functions

- So far, we've seen many different types of functions:
 - Our programs comprise a single function called `main()`.
 - Built-in Python functions (`abs`, `range`, `input`...)
 - Functions from the standard libraries (`math.sqrt`)

Coming up: Functions, Informally

5

Functions, Informally

- A function is like a *subprogram*, a small program inside of a program.
- The basic idea – we write a sequence of statements and then give that sequence a name (*define* a function).
- We can then execute this sequence at any time by referring to the name. (*invoke* or *call* a function)

Coming up: Coolness Calculator

6

Coolness Calculator

```
def main():
    johnPythonSkill = 10
    johnMontyPythonTrivia = 20

    johnCoolness = (johnPythonSkill * 2) + \
        (johnMontyPythonTriviaScore * 1.5)

    if johnCoolness > 30:
        print 'I will ask John out'
    elif johnCoolness > 20:
        print 'I will set him up with my friend Mary'
    else:
        print 'I will send John a Monty Python DVD and',
        'CS112 textbook.'
```

Works great for John, but I have other people to check!

Coming up: Making a function

7

Making a function

- Calculating Coolness

```
def main():
    johnPythonSkill = 10
    johnMontyPythonTrivia = 20

    johnCoolness = (johnPythonSkill * 2) + \
        (johnMontyPythonTriviaScore * 1.5)

    if johnCoolness > 30:
        print 'I will ask John out'
    elif johnCoolness > 20:
        print 'I will set him up with my friend Mary'
    else:
        print 'I will send John a Monty Python DVD and',
        'CS112 textbook.'
```

Make this a function in case our coolness definition changes in the future (python * 10?)

Coming up: What can change?

8

What can change?

• Calculating Coolness

```
johnCoolness = (johnPythonSkill * 2) + \
               (johnMontyPythonTriviaScore * 1.5)
```

- Determine what you think may change from person to person and make those parameters
- PythonSkill
- PythonTriviaScore
- PythonSkillWeight (maybe)
- PythonTriviaWeight (maybe)

Coming up: Try #1

9

Try #1

```
def calculateCoolness():
    johnCoolness = (johnPythonSkill * 2) + \
                   (johnMontyPythonTriviaScore * 1.5)

def main():
    johnPythonSkill = 10
    johnMontyPythonTrivia = 20

    calculateCoolness()
    if johnCoolness > 30:
        print 'I will ask John out'
    elif johnCoolness > 20:
        print 'I will set him up with my friend Mary'
    else:
        print 'I will send John a Monty Python DVD and',
        'CS112 textbook.'
```

This does not work because of variable scope!

Coming up: Variable Scope

10

Variable Scope



- Every variable has a “scope”.
- The *scope* of a variable refers to the places in a program a given variable can be referenced.
- **Variables defined in a function are local variables** and can only be referenced directly in that function

Coming up: Try #2

11

Try #2

```
def calculateCoolness(johnPythonSkill, johnMontyPythonTrivia):
    johnCoolness = (johnPythonSkill * 2) + \
                   (johnMontyPythonTriviaScore * 1.5)

def main():
    johnPythonSkill = 10
    johnMontyPythonTrivia = 20

    calculateCoolness(johnPythonSkill, johnMontyPythonTrivia)
    if johnCoolness > 30:
        print 'I will ask John out'
    elif johnCoolness > 20:
        print 'I will set him up with my friend Mary'
    else:
        print 'I will send John a Monty Python DVD and',
        'CS112 textbook.'
```

Adding parameters makes things better.. But still a problem!
johnCoolness is local in calculateCoolness... how to fix?

Coming up: Try #3

12

Try #3

```
def calculateCoolness(johnPythonSkill, johnMontyPythonTrivia, johnCoolness):
    johnCoolness = (johnPythonSkill * 2) + \
        (johnMontyPythonTriviaScore * 1.5)

def main():
    johnPythonSkill = 10
    johnMontyPythonTrivia = 20
    johnCoolness = 0

    calculateCoolness(johnPythonSkill, johnMontyPythonTrivia, \
        johnCoolness)
    if johnCoolness > 30:
        print 'I will ask John out'
    elif johnCoolness > 20:
        print 'I will set him up with my friend Mary'
    else:
        print 'I will send John a Monty Python DVD and',
        ' CS112 textbook. '
```

Seems right, but Python uses copies (pass by value)... so this also does not work!

Coming up: Try #4

13

Try #4

```
def calculateCoolness(johnPythonSkill, johnMontyPythonTrivia):
    johnCoolness = (johnPythonSkill * 2) + \
        (johnMontyPythonTriviaScore * 1.5)
    return johnCoolness

def main():
    johnPythonSkill = 10
    johnMontyPythonTrivia = 20
    johnCoolness = 0

    johnCoolness = calculateCoolness(johnPythonSkill,
    johnMontyPythonTrivia)
    if johnCoolness > 30:
        print 'I will ask John out'
    elif johnCoolness > 20:
        print 'I will set him up with my friend Mary'
    else:
        print 'I will send John a Monty Python DVD and',
        ' CS112 textbook. '
```

Add a return value to get information out of a function! This works... but variables should be generically named

Coming up: Try #5

14

Try #5

```
def calculateCoolness( pythonSkill, montyPythonTrivia):
    coolness = ( pythonSkill * 2) + \
        ( montyPythonTriviaScore * 1.5)
    return coolness

def main():
    name = raw_input("Who are we checking? ")
    pythonSkill = input("What is their Python skill?")
    montyPythonTrivia = input("What is their trivia score?")
    coolness = 0
    coolness = calculateCoolness( pythonSkill, montyPythonTrivia)
    if coolness > 30:
        print 'I will ask ',name,' out'
    elif coolness > 20:
        print 'I will set him up with my friend Mary'
    else:
        print 'I will send ',name,' a Monty Python DVD and',
        ' CS112 textbook. '
```

Now our coolness detector can tell us who we should date... whew, much easier than the non-Python way!

Coming up: Functions can call other functions

15

Functions can call other functions

Any function can call any other function in your module

```
def func1(from):
    print "I am in func 1 from", from

def func2():
    print "I am in func 2"
    for i in range(3):
        func1("f2")

def main():
    func2()
    func1("main")
```

Output:

```
I am in func2
* I am in func1 from f2
* I am in func1 from f2
* I am in func1 from f2
* I am in func1 from main
```

Coming up: Function Lifecycle

16

Function Lifecycle

Recall:

We are formal parameters

```
def function1(formalParameter1, fp2, fp3):  
    # Do something  
    return someVal
```

```
def main():  
    answer = \  
        function1(actualParameter1, ap2, ap3)
```

We are actual parameters

Function Call Lifecycle

1. main is suspended
2. formal parameters are assigned values from actual parameters
3. function body executes
4. left-hand-side of function call is assigned value of whatever is *returned* from function
5. Control returns to the point just after where the function was called.

Coming up: Functions and Parameters: The Details

17

Functions and Parameters: The Details

- Each function is its own little subprogram. The variables used inside of one function are *local* to that function, even if they happen to have the same name as variables that appear inside of another function.
- The only way for a function to see a variable from another function is for that variable to be passed as a parameter.
- The *scope* of a variable refers to the places in a program a given variable can be referenced.

Coming up: Functions and Parameters: The Details

18

Functions and Parameters: The Details

- Formal parameters, like all variables used in the function, are only accessible in the body of the function.
- Variables with identical names elsewhere in the program are distinct from the formal parameters and variables inside of the function body.

Coming up: Trace through some code

19

Trace through some code

```
def main():  
    sing("Fred")  
    print  
    sing("Lucy")  
  
def sing(person):  
    happy()  
    print "Happy birthday, dear", person + "."  
    happy()  
  
person = "Fred"
```

Note that the variable `person` has just been initialized.

Coming up: Trace through some code

20

Trace through some code

- At this point, Python begins executing the body of `sing`.
- The first statement is another function call, to `happy`. What happens next?
- Python suspends the execution of `sing` and transfers control to `happy`.
- `happy` consists of a single print, which is executed and control returns to where it left off in `sing`.

Coming up: Trace through some code

21

Trace through some code

```
def main():
    person = "Fred"
    sing("Fred")
    print
    sing("Lucy")

def sing(person):
    happy()
    happy()
    print "Happy birthday, dear", person + "."
    happy()

def happy():
    print "Happy Birthday to you!"

person: "Fred"
```

- Execution continues in this way with two more trips to `happy`.
- When Python gets to the end of `sing`, control returns to `main` and continues immediately following the function call.

Coming up: Trace through some code

22

Trace through some code

```
def main():
    person = "Fred"
    sing("Fred")
    print
    sing("Lucy")

def sing(person):
    happy()
    happy()
    print "Happy birthday, dear", person + "."
    happy()
```

- Notice that the `person` variable in `sing` has disappeared!
- The memory occupied by local function variables is reclaimed when the function exits.
- Local variables do **not** retain any values from one function execution to the next.

Coming up: Trace through some code

23

Trace through some code

```
def main():
    sing("Fred")
    print
    sing("Lucy")

def sing(person):
    happy()
    happy()
    print "Happy birthday, dear", person + "."
    happy()

person: "Lucy"
```

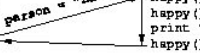
- The body of `sing` is executed for `Lucy` with its three side trips to `happy` and control returns to `main`.

Coming up: Trace through some code

24

Trace through some code

```
def main():  
    sing('Fred')  
    print  
    sing('Lucy')  
def sing(person):  
    happy()  
    happy()  
    print "Happy birthday, dear", person + "."  
    happy()
```



Coming up: Trace through some code

25

Trace through some code

- One thing not addressed in this example was multiple parameters. In this case the formal and actual parameters are matched up based on *position*, e.g. the first actual parameter is assigned to the first formal parameter, the second actual parameter is assigned to the second formal parameter, etc.

Coming up: Trace through some code

26

Trace through some code

- As an example, consider the call to drawBar:
`drawBar(win, 0, principal)`
- When control is passed to drawBar, these parameters are matched up to the formal parameters in the function heading:
`def drawBar(window, year, height):`

Coming up: Functions and Parameters: The Details

27

Functions and Parameters: The Details

- The net effect is as if the function body had been prefaced with three assignment statements:

```
window = win  
year = 0  
height = principal
```

Coming up: Parameters are INPUT to a function

28

Parameters are INPUT to a function

- Passing parameters provides a mechanism for initializing the variables in a function.
- Parameters act as inputs to a function.
- We can call a function many times and get different results by changing its parameters.

Coming up: Return values are OUTPUT from a function

29

Return values are OUTPUT from a function

- We've already seen numerous examples of functions that return values to the caller.

```
discRt = math.sqrt(b*b - 4*a*c)
```

- The value $b*b - 4*a*c$ is the actual parameter of `math.sqrt`.
- We say `sqrt` *returns* the square root of its argument.

Coming up: Functions That Return Values

30

Functions That Return Values

- This function returns the square of a number:

```
def square(x):  
    return x*x
```

- When Python encounters `return`, it exits the function and returns control to the point where the function was called.
- In addition, the value(s) provided in the `return` statement are sent back to the caller as an expression result.

Coming up: Return examples

31

Return examples

- ```
>>> square(3)
9
```
- ```
>>> print square(4)  
16
```
- ```
>>> x = 5
>>> y = square(x)
>>> print y
25
```
- ```
>>> print square(x) + square(3)  
34
```

Coming up: Multiple Return values

32

Multiple Return values

- Sometimes a function needs to return more than one value.
- To do this, simply list more than one expression in the `return` statement.
- ```
def sumDiff(x, y):
 sum = x + y
 diff = x - y
 return sum, diff
```

Coming up: Multiple Return Values

33

## Multiple Return Values

- When calling this function, use simultaneous assignment.
- ```
num1, num2 = input("Please enter two numbers (num1, num2) ")  
s, d = sumDiff(num1, num2)  
print "The sum is", s, "and the difference is", d
```
- As before, the values are assigned based on position, so `s` gets the first value returned (the sum), and `d` gets the second (the difference).

Coming up: Secretly -- all functions return a value

34

Secretly -- all functions return a value

- One "gotcha" -- all Python functions return a value, whether they contain a `return` statement or not. Functions without a `return` hand back a special object, denoted `None`.
- A common problem is writing a value-returning function and omitting the `return`!
- Watch out!

Coming up: Python passes parameter values

35

Python passes parameter values

```
def addToVar(x):  
    x = x + 2  
  
def main():  
    x = 10  
    addToVar(x)  
    print x
```

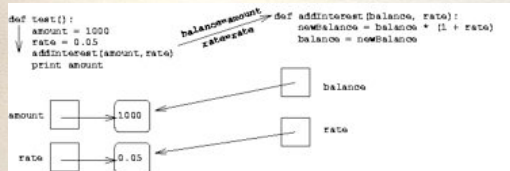
Output:
10

Why? Python passes copies of the value, so changing the copy doesn't do anything!!
(This is called "pass by value")

Coming up: Lets explain

36

Lets explain



Picture all variables as arrows pointing to a number, changing the variable's value just makes the arrow point somewhere else

Coming up: Pass by value

37

Pass by value

- Executing the first line of `addInterest` creates a new variable, `newBalance`.
- `balance` is then assigned the value of `newBalance`.

```

def addInterest(balance, rate):
    newBalance = balance * (1 + rate)
    balance = newBalance

def test():
    amount = 1000
    rate = 0.05
    addInterest(amount, rate)
    print amount
  
```

Coming up: Pass by value

38

Pass by value

- `balance` now refers to the same value as `newBalance`, but this had no effect on `amount` in the `test` function.

```

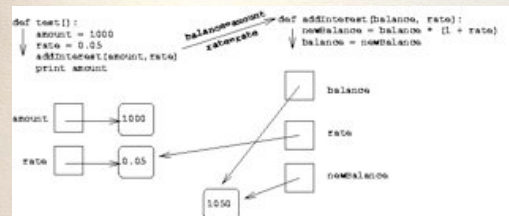
def addInterest(balance, rate):
    newBalance = balance * (1 + rate)
    balance = newBalance

def test():
    amount = 1000
    rate = 0.05
    addInterest(amount, rate)
    print amount
  
```

Coming up: Pass by value

39

Pass by value



Coming up: Pass by value

40

Pass by value

- Execution of `addInterest` has completed and control returns to `test`.
- The local variables, including the parameters, in `addInterest` go away, but `amount` and `rate` in the `test` function still refer to their initial values!

```
def addInterest(balance, rate):  
    newBalance = balance * (1 +  
        rate)  
    balance = newBalance  
  
def test():  
    amount = 1000  
    rate = 0.05  
    addInterest(amount, rate)  
    print amount
```

Coming up: Pass by value

41

Pass by value

- To summarize: the formal parameters of a function only receive the *values* of the actual parameters. The function does not have access to the variable that holds the actual parameter.
- Python is said to pass all parameters *by value*.

Coming up: But...

42

But...

```
def func1(input):  
    for i in range(3):  
        input[i] = input[i] + 10
```

```
def main():  
    myList = [1, 2, 3]  
    func1(myList)  
    print myList
```

Output:
11, 12, 13

Why why
why?

Coming up: Answers

43

Answers

- A. Python is just messed up
- B. Mr. Fleck lied to us and some things are not passed by value
- C. Who cares, I'm going to change to a history major.. Python annoys me now
- D. Something different happens with mutable data types

Coming up: Lets look at this code

44

Lets look at this code

```
# addinterest3.py
# Illustrates modification of a mutable parameter (a list).

def addInterest(balances, rate):
    for i in range(len(balances)):
        balances[i] = balances[i] * (1+rate)

def test():
    amounts = [1000, 2200, 800, 360]
    rate = 0.05
    addInterest(amounts, 0.05)
    print amounts

test()
```

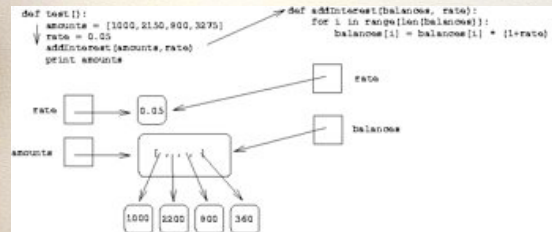
Output:

[1050.0, 2310.0, 840.0, 378.0]

Coming up: The truth ... really this time!

45

The truth ... really this time!



The "value" of a list is a group of arrows

Coming up: The truth ... really this time!

46

The truth ... really this time!

- Next, `addInterest` executes. The loop goes through each index in the range 0, 1, ..., length-1 and updates that value in `balances`.

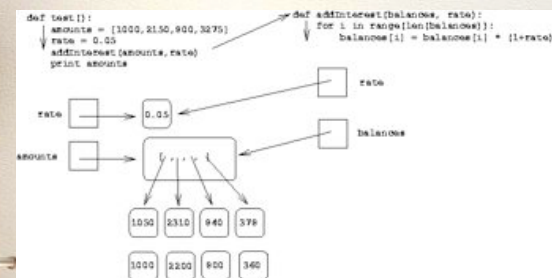
```
def addInterest(balances, rate):
    for i in range(len(balances)):
        balances[i] = balances[i] * (1+rate)

def test():
    amounts = [1000, 2200, 800, 360]
    rate = 0.05
    addInterest(amounts, 0.05)
    print amounts
```

Coming up: The truth ... really this time!

47

The truth ... really this time!



Coming up: The truth ... really this time!

48

The truth ... really this time!

- In the diagram the old values are left hanging around to emphasize that the numbers in the boxes have not changed, but the new values were created and assigned into the list.
- The old values will be destroyed during garbage collection.

```
def addInterest(balances, rate):  
    for i in range(len(balances)):  
        balances[i] = balances[i]  
        * (1+rate)  
  
def test():  
    amounts = [1000, 2200, 800,  
360]  
    rate = 0.05  
    addInterest(amounts, 0.05)  
    print amounts
```

Coming up: The final answer

49

The final answer

- When `addInterest` terminates, the list stored in `amounts` now contains the new values.
- The variable `amounts` wasn't changed (it's still a list), but the state of that list has changed, and this change is visible to the calling program.
- So... the final answer is, we did NOT change the value of the list, we changed where the list arrows (inside the list) pointed (and Mr. Fleck is not a liar... just goofy, and a bit crazy...)

Coming up: One last time for the cheap seats...

50

One last time for the cheap seats...

- Parameters are always passed by value. However, if the value of the variable is a mutable object (like a list), then changes to the state of the object *will* be visible to the calling program.

Coming up: If your brain hurts...

51

If your brain hurts...



Coming up: Lets write a Hangman Game

52

Lets write a Hangman Game

- When you write a game you first can decide what are the core functions and variables we need.
- Let think of Hangman... what I want it to look like is this:

Guesses: s, q, r, e t

Current word: __ t _ o n

Enter guess or 1 to quit ->

What information
(variables) do I need
to know to generate
this?

Coming up: Hangman State

53

Hangman State

Hangman
Drawn Here

Guesses: s, q, r, e t

Current word: __ t _ o n

Enter guess or 1 to quit ->

What information
(variables) do I need
to know to generate
this?

Coming up: Hangman State

54

Hangman State

- `misses = 0` # How many bad guesses have they had?
- `lettersGuessed = []` # Empty list of the letters already guessed
- `wordToGuess = "python"` # Should ask the user for this
- Got it... let's move to the pseudocode

Coming up: Hangman Pseudocode

55

Hangman Pseudocode

- What is it?

Coming up: Hangman Pseudocode

56

Hangman Pseudocode

- print the hangman
- print the word
- ask the user for input
 - check if the letter was already used (if so, warn the user and start over at step 1)
 - update the list of used letters
 - check if the letter is in the word
 - if so, check if the user has won
 - if not, check if the user has lost
- Not bad.. on to hangman.py!

Coming up: Default Parameters

57

Default Parameters

Function parameters can have default values

```
def exponent(num, exp=2):  
    return num ** exp
```

```
print exponent(4)  
print exponent(4, 3)
```

Output:
16
64

Coming up: Default Parameters

58

Default Parameters

Function parameters can have default values

```
def exponent(num, exp=2):  
    return num ** exp
```

```
print exponent(4, 3, 45)
```

Output:
TypeError: exponent() takes
at most 2 arguments (3 given)

Coming up: Default Parameters

59

Default Parameters

Non-default arguments cannot follow default arguments!

```
def exponent(num, exp=2, temp):  
    return num ** exp
```

```
print exponent(4, 3, 45)
```



Coming up: Default Parameters

60

Default Parameters

Arguments are assigned left to right

```
def exponent(num, exp=2, temp=5):  
    print "Num:%d Exp:%d Temp:%d" \  
        %(num, exp, temp)  
    return num ** exp
```

```
print exponent(4)  
print exponent(4, 3)  
print exponent(4, 3, 45)
```

Output:
Num:4 Exp:2 Temp:5
16
Num:4 Exp:3 Temp:5
64
Num:4 Exp:3 Temp:45
64

Coming up: Named Arguments

61

Named Arguments

What if you want to use the default for parameter 2, but give a value for parameter 3?

```
def exponent(num, exp=2, temp=5):  
    print "Num:%d Exp:%d Temp:%d" \  
        %(num, exp, temp)  
    return num ** exp
```

```
exponent(34, temp=3)  
exponent(temp=3, num=34)
```

Output:
Num:34 Exp:2 Temp:3
Num:34 Exp:2 Temp:3

More info at: <http://www.python.org/doc/current/tut/node6.html>

Coming up: What prints?

62

What prints?

```
def myFunction(a, b, c=4, d=5):  
    print a, b, c, d
```

```
myFunction(5, 4, 3, 2)  
myFunction(3, 4)  
myFunction(3, 4, 5)  
myFunction(3, 4, d=5)  
myFunction(b=1, a=2)
```

5 4 3 2

3 4 4 5

3 4 5 5

3 4 4 5

2 1 4 5

Coming up: What prints?

63