Python Programming: Introduction to Computer Science

Functions,Variables,Modules

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.

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)
```


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()
```

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, raw_input…)
  - Functions from the standard libraries (math.sqrt)

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)

Parameters

- Functions can accept data from other functions as input.

```
def printHello(name):
    print "Hello", name

def main():
    aVariable = "Mary"
    printHello("John")
    printHello("Carl")
    printHello(aVariable)
```

When the function is called, the parameter "name" is assigned the value from the caller.

A parameter is just a variable that gets reassigned everytime someone calls the function

Parameter Terminology

```
def printHello(name):
    print "Hello", name

def main():
    aVariable = "Mary"
    printHello("John")
    printHello("Carl")
    printHello(aVariable)
```

Name is a formal parameter

When the function is called, the formal parameter "name" is assigned the value of the actual parameter.

A parameter is just a variable that gets reassigned everytime someone calls the function

John, Carl and aVariable are actual parameters

Multiple Parameters

```
def parents(mom, dad):
    print "My parents are ", mom," and ", dad

def main():
    parents("Wilma", "Fred")
    parents("Homer", "Marge") # Oops… order counts!
```


Returning information from a function

```
def getTax(cost):
    tax = cost * 0.08
    return tax

def main():
    carTax = getTax(25000)
    burgerTax = getTax(2.99)
    taxOnTax = getTax(carTax)
```

Return values enable functions to process data and return new information

To use a return value you must assign the function call as the RHS of an assignment statement

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!

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'
    if johnCoolness > 20:
        print 'I will set him up with my friend Mary'

        t 'I will send John a Monty Python DVD and',
```

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

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)

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!

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 (or used).
- **Variables defined in a function are local variables** and can only be referenced directly in that function

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?

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!

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

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!

Functions Part II

Advanced concepts

Reminders

```
def addTwoNumbers(num1, num2):
    sum = num1 + num2
    print "The sum is %d " %(sum)
    return sum

def main():
    sum = addTwoNumbers(12,34)
```

Formal Parameters

Formal parameters get replaced with actual parameters from function call.

Return value

Actual Parameters

Execute (call) the function. Think of this as the function runs and then you replace the call with the return value

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
```


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.

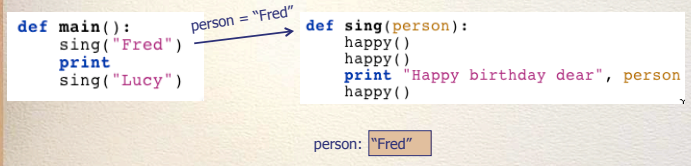
Functions and Parameters: The Details

- 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.

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.

Trace through some code

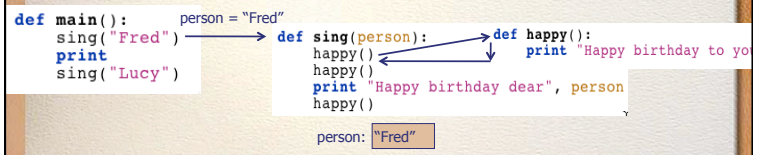


Note that the variable `person` has just been initialized to a value. What value?

Trace through some code

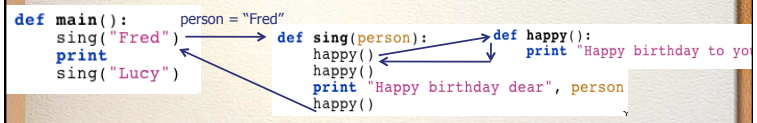
- At this point, Python begins executing the body of `sing`.
- The first statement is another function call, to `happy`. What happens next?
- Python suspends the execution of `sing` and transfers control to `happy`.
- `happy` consists of a single print, which is executed and control returns to where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

Trace through some code



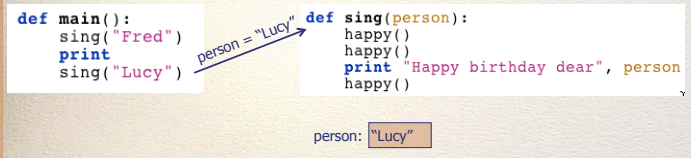
- Execution continues in this way with two more trips to `happy`.
- When Python gets to the end of `sing`, control returns to `main` and continues immediately following the function call.

Trace through some code



- 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.

Trace through some code



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

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.

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):
```

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
```


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.

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 expression `b*b - 4*a*c` is the actual parameter of `math.sqrt`.
- We say `sqrt` *returns* the square root of its argument.
- You must assign the function to a variable in order to save the result of the function

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.

Return statement exit the function immediately. Only use them when you want to exit the function.
If you return from the main function what typically happens?

Return examples

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

Picture in your head replacing the call: square(x) with it's return value 25, so:

y = ~~square(x)~~ 25

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 separated by commas.

```
def sumDiff(x, y):
    sum = x + y
    diff = x – y
    return sum, diff
```


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).

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!

Function Libraries

- Functions can be defined in one file and used in another. These are function libraries
- Like the math library. To use it:
  - import math
  - val = math.sqrt(300)
    # Note: you preface the function defined in math.py with “math” (the filename)

Create our own function library

- Lets create a function library with function to get the area and perimeter of a square.

Python passes parameter values

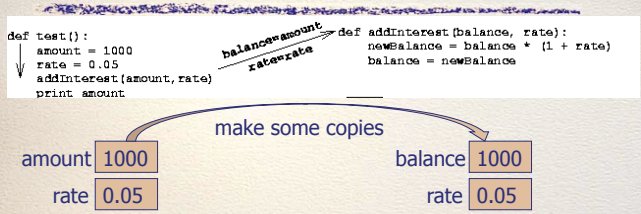
```
def addToVar(x):
    x = x + 2

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”)

Lets explain



Picture all variables as boxes with a data in them. Calling a function simply makes a copy of the box.

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
```


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
```


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
```

Changing rate in addInterest, what changes in test?

Pass by value

- To summarize: the formal parameters of a function only receive copies of the *values* of the actual parameters.
- Python is said to pass all parameters *by value*.

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? Anger rising….

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

Mutable types hold addreses

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

aList | You'll find the data at 196 RAM Lane

Your computer's memory (RAM)

196 RAM Lane	0x200 RAM Court	0x210 RAM Blvd
1	abc	1
2	"SecretPassword"	2
3	563	3

Is this what people mean by "a walk down memory lane?"

Mutable types hold addreses

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

```
def func1(someList):
    someList[1] = 23
```

make some copies

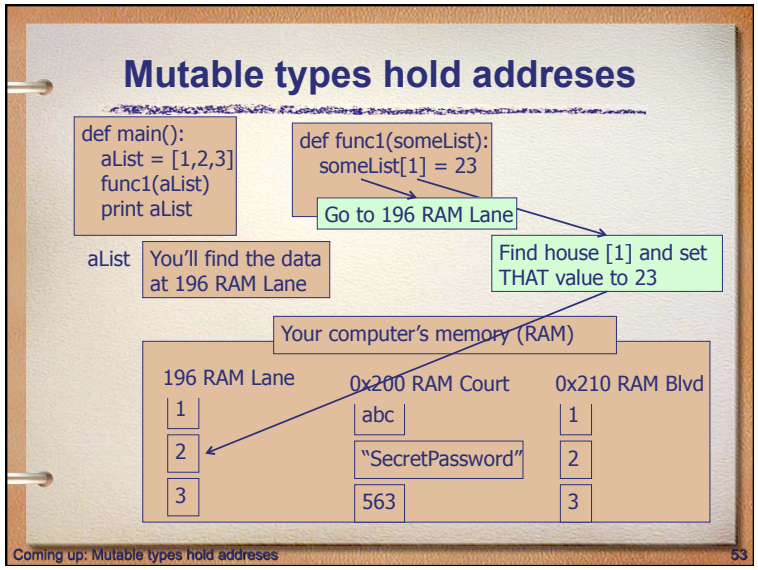
aList | You'll find the data at 196 RAM Lane

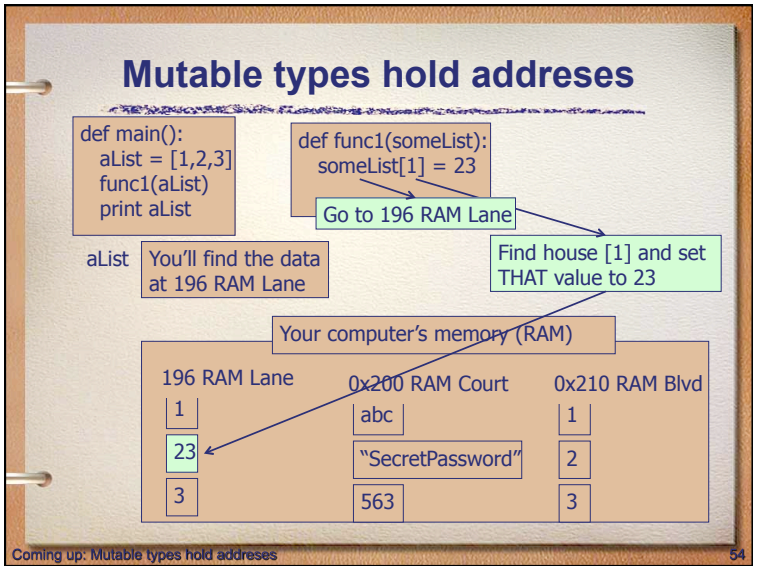
someList | You'll find the data at 196 RAM Lane

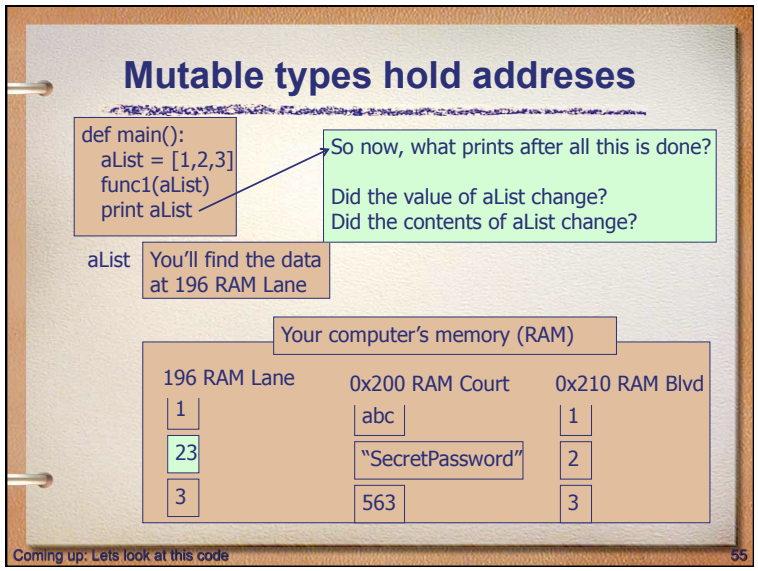
Your computer's memory (RAM)

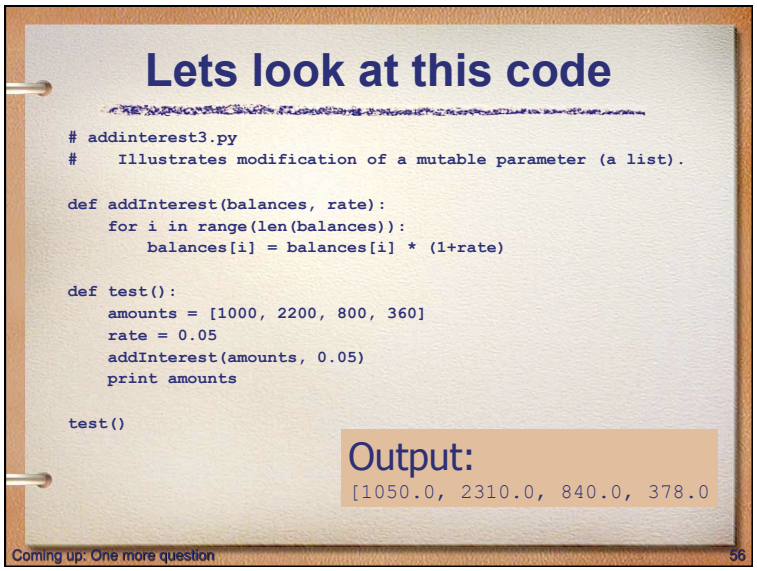
196 RAM Lane	0x200 RAM Court	0x210 RAM Blvd
1	abc	1
2	"SecretPassword"	2
3	563	3

Is this what people mean by "a walk down memory lane?"


Mutable types hold addreses
def main():
aList = [1,2,3]
func1(aList)
print aList
def func1(someList):
someList[1] = 23
Go to 196 RAM Lane
aList
You'll find the data at 196 RAM Lane
Find house [1] and set THAT value to 23
Your computer's memory (RAM)
196 RAM Lane
0x200 RAM Court
0x210 RAM Blvd
1
2
3
abc
"SecretPassword"
563
1
2
3
Coming up: Mutable types hold addreses
53


Mutable types hold addreses
def main():
aList = [1,2,3]
func1(aList)
print aList
def func1(someList):
someList[1] = 23
Go to 196 RAM Lane
aList
You'll find the data at 196 RAM Lane
Find house [1] and set THAT value to 23
Your computer's memory (RAM)
196 RAM Lane
0x200 RAM Court
0x210 RAM Blvd
1
23
3
abc
"SecretPassword"
563
1
2
3
Coming up: Mutable types hold addreses
54


Mutable types hold addreses
def main():
aList = [1,2,3]
func1(aList)
print aList
So now, what prints after all this is done?
Did the value of aList change?
Did the contents of aList change?
aList
You'll find the data at 196 RAM Lane
Your computer's memory (RAM)
196 RAM Lane
0x200 RAM Court
0x210 RAM Blvd
1
23
3
abc
"SecretPassword"
563
1
2
3
Coming up: Lets look at this code
55


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: One more question
56

One more question

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

def addInterest(balances, rate):
    balances = [1,2,8,3] # Oops! - Bernie Madoff
    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()
```


One more question

```
balances = [1,2,8,3] # Oops! - Bernie Madoff
```

balances → You'll find the data at 555 RAM Court

balances → You'll find the data at 898 RAM Way

Very different from...

```
balances[3] = 27
```

balances → You'll find the data at 555 RAM Court

```
- Go to address 555 RAM Court
- Change house [3] to hold 27
```

The variable "balances" holds the address. When you index/slice it you are going to the address.

When you change address held, you are changing where the variable data is found

The final answer

- When you call a function you are always passing a copy of the data.
- For immutable types the copy is the data, and thus changing the copy does not affect the original value (in the original function)
- For mutable types you are passing an address and if you change the contents of the data, the changes will show up in the original function

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.

A Note on Globals

- Now that you know how to use parameters and return values you should avoid global variables.
- Globals should be used sparingly
- One appropriate use is for constant values you need throughout your code (like PI or "SALES_TAX")
- When defining global constants a coding convention is to use all capital letters for the variable name.

Types of Arguments

Three types of function arguments can be used

- Positional Arguments
- Default Arguments
- Keyword Arguments

- See examples: samplecode/functions/ArgumentTypeExamples.py

Positional Arguments

- Value of the formal parameter is set based on the position (left to right):
  - def func(sum, label):
    …
  - func(22,"Carl") # Call the function
  - # Values are assigned by position so: sum=22 and label="Carl" when executing func
  - func(22) # Error – not enough arguments!

Default Arguments

- Value of the formal parameter is set based on the position, or if not present based on default:
  - def func(sum=99, label="Unknown"):
    …
  - func(22,"Carl") # Call the function
  - # Values are assigned by position so: sum=22 and label="Carl" when executing func
  - func(22) # Okay! sum=22, and label=Unknown (default value)

Default Arguments

- def func(sum=99, label="Unknown"):
  …
- func(22,"Carl")
  - # sum=22, label="Carl"
- func(22)
  - # Okay! sum=22, and label=Unknown (default value)
  - Arguments assigned left to right.
- func()
  - Okay, sum=99, label="Unknown"
- func("Carl")
  - # Also okay, but maybe not what you want:
  - sum="Carl", label="Unknown" --- order matters!

Default Arguments

- def func(sum=99, label="Unknown"):
  …
- func("Carl")
  - # Also okay, but maybe not what you want:
  - sum="Carl", label="Unknown" --- order matters!
- When defining a function, all default arguments must be at the END of the parameter list
  - def func(sum=99, label): # ERROR – Non-default argument follows default argument.
- To do something like this use keyword arguments

Keyword Arguments

Specify argument name during function call, then position doesn't matter

- def func(sum, label):

- func(sum=99, label="A label")
- func(label="A label", sum=99)

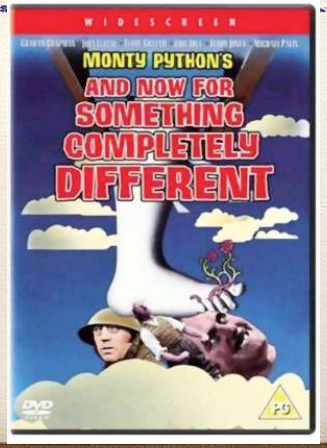
- func(label="Bob") # Does this work?

What prints?

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

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

If your brain hurts…



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?

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?

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

Hangman Pseudocode

- What is it?

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!