Special Topics in Computer Science- CSC 4992

Overview of Functions

Python Functions

- There are two kinds of functions in Python.
  - Built-in functions that are provided as part of Python input(), type(), float(), int() ...
  - Functions that we define ourselves and then use
- We treat the of the built-in function names as "new" reserved words (i.e. we avoid them as variable names)

What Is a Function?

A function is a chunk of code that can be called by name wherever we want to run that code

```
def sqr(n):                   # Definition
    return n ** 2



...

print(sqr(2))                  # Call: Displays 4

print(sqr(33))                 # Call: Displays 1089

print(sqr(etc))                # Call: Displays whatever
```

Using Functions: Combination

Functions can be used to compute values, wherever operand expressions are expected

```
a = sqr(4)
b = sqr(3)
c = math.sqrt(a + b)
```

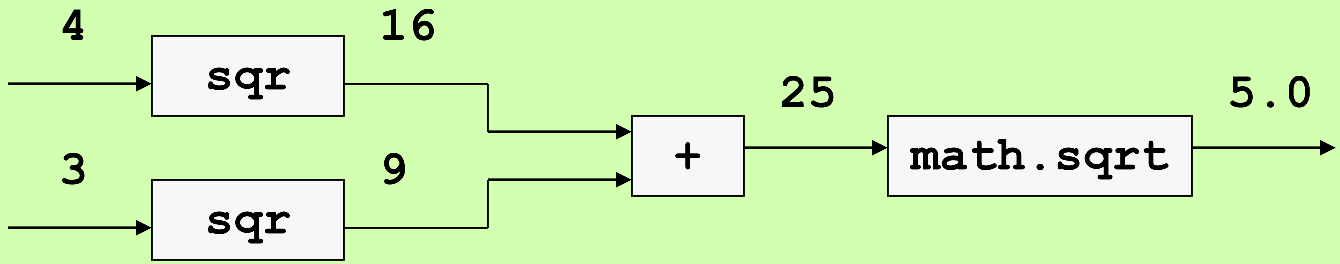
Using Functions: Combination

Functions can be used to compute values, wherever operand expressions are expected

```
a = sqr(4)
b = sqr(3)
c = math.sqrt(a + b)

# Or use function calls as operands:

c = math.sqrt(sqr(4) + sqr(3))
```



Arguments and Return Values

- A function can receive data from its caller (arguments)

- A function can return a single value to its caller

```
y = math.sqrt(2)
```

Programmer-Defined Functions

- A function allows the programmer to define a general algorithm in one place and use it in many other places (avoid repetitive patterns)

- A function replaces many lines of code with a single name (abstraction principle)

Function Definition Syntax: Parameters and `return` Statements

The *function header* includes 0 or more *parameter* names

```
def sqr(n):                # Definition
    return n * n
```

```
def <function name>(<param name>, …, <param name>):
    <sequence of statements>
```

The **`return`** statement exits the function call with a value

A General Input Function

Define a function that obtains a valid input number from the user

The function expects a string prompt and the lower and upper bounds of the range of valid numbers as arguments

The function continues to take inputs until a valid number is entered; if an invalid number is entered, the function prints an error message

The function returns the valid number

Example Use

Pretend that the function has already been defined and imagine its intended use

```
>>> rate = getValidNumber("Enter the rate: ", 1, 100)
Enter the rate: 120
Error: the number must range from 1 through 100
Enter the rate: 99
>>> rate
99
```

```
>>> size = getValidNumber("Enter the size: ", 1, 10)
Enter the size: 15
Error: the number must range from 1 through 10
Enter the size: 5
>>> size
5
```

Definition

```
def getValidNumber(prompt, lower, upper):
    """Repeatedly inputs a number until that
    number is within the given range."""
```

A function definition can include a docstring

`help(getValidNumber)` displays this information

Definition

```
def getValidNumber(prompt, lower, upper):
    """Repeatedly inputs a number until that
    number is within the given range."""
    while True:
        number = int(input(prompt))
        if number < lower or number > upper:
            print("Error: the number must range from " + \
                  str(lower) + " through " + str(upper))
        else:
            return number
```

The **`return`** statement exits both the loop and the function call

The **`\`** symbol is used to break a line of Python code

Return Values

- Often a function will take its arguments, do some computation and return a value to be used as the value of the function call in the calling expression. The return keyword is used for this.

```
def greet():    return "Hello "

print (greet(), "Glenn")
print (greet(), "Sally")
```

```
Hello Glenn
Hello Sally
```

Data Encryption Revisited

```
>>> print(encrypt("Exam Friday!"))
69 120 97 109 32 70 114 105 100 97 121 33
```

```
def encrypt(source):
    """Builds and returns an encrypted version of
    the source string."""
    code = ""
    for ch in source:
        code = code + str(ord(ch)) + " "
    return code
```

`source` is a *parameter* and **`code`** and **`ch`** are *temporary variables*

They are visible only within the body of the function

Data Decryption Revisited

```
>>> print(decrypt(encrypt("Exam Friday!")))
Exam Friday!
```

```
def decrypt(code):
    """Builds and returns a decrypted version of
    the code string."""
    source = ""
    for word in code.split():
        source = source + chr(int(word))
    return source
```

Functions that Modify Parameters

- Return values are the main way to send information from a function back to the caller.
- Sometimes, we can communicate back to the caller by making changes to the function parameters.
- Understanding when and how this is possible requires the mastery of some subtle details about how assignment works and the relationship between actual and formal parameters.

Functions that Modify Parameters

- Suppose you are writing a program that manages bank accounts. One function we would need to do is to accumulate interest on the account. Let's look at a first-cut at the function.

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

Functions that Modify Parameters

- The intent is to set the balance of the account to a new value that includes the interest amount.
- Let's write a main program to test this:

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

Functions that Modify Parameters

- We hope that that the 5% will be added to the amount, returning 1050.
- ```
  >>> test()
  1000
  ```
- What went wrong? Nothing!
  ```

Functions that Modify Parameters

- The first two lines of the test function create two local variables called amount and rate which are given the initial values of 1000 and 0.05, respectively.

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

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

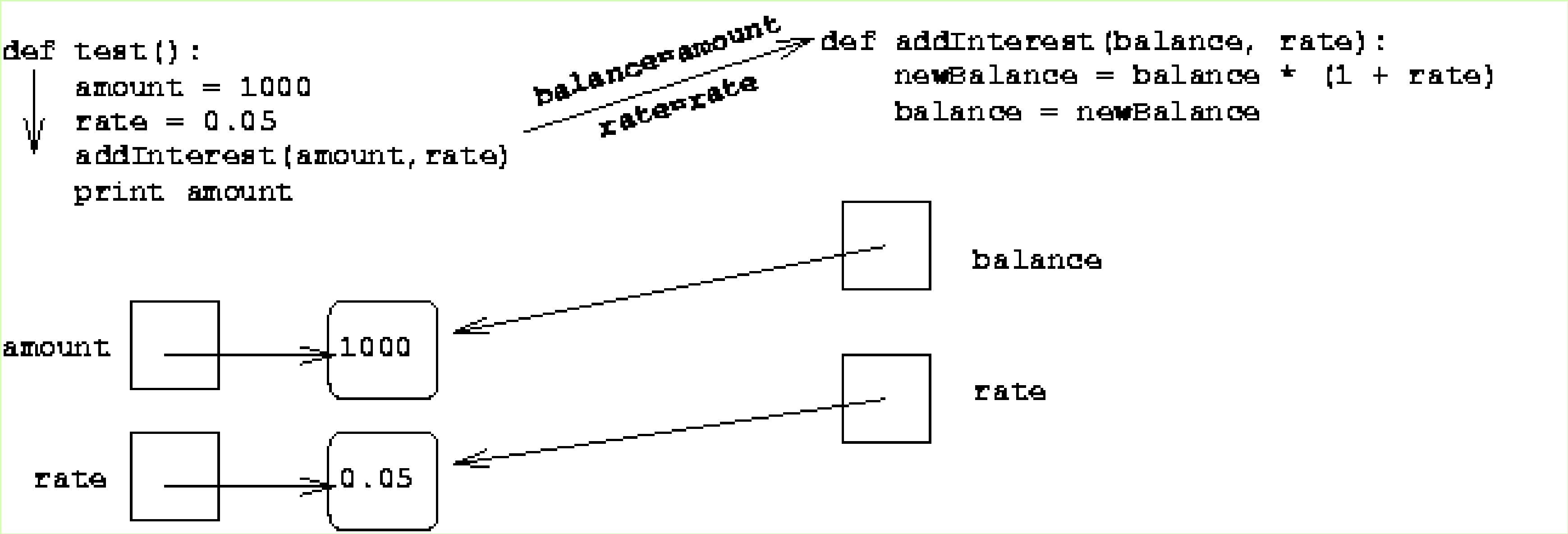
Functions that Modify Parameters

- Control then transfers to the addInterest function.
- The formal parameters balance and rate are assigned the values of the actual parameters amount and rate.
- Even though rate appears in both, they are separate variables (because of scope rules).

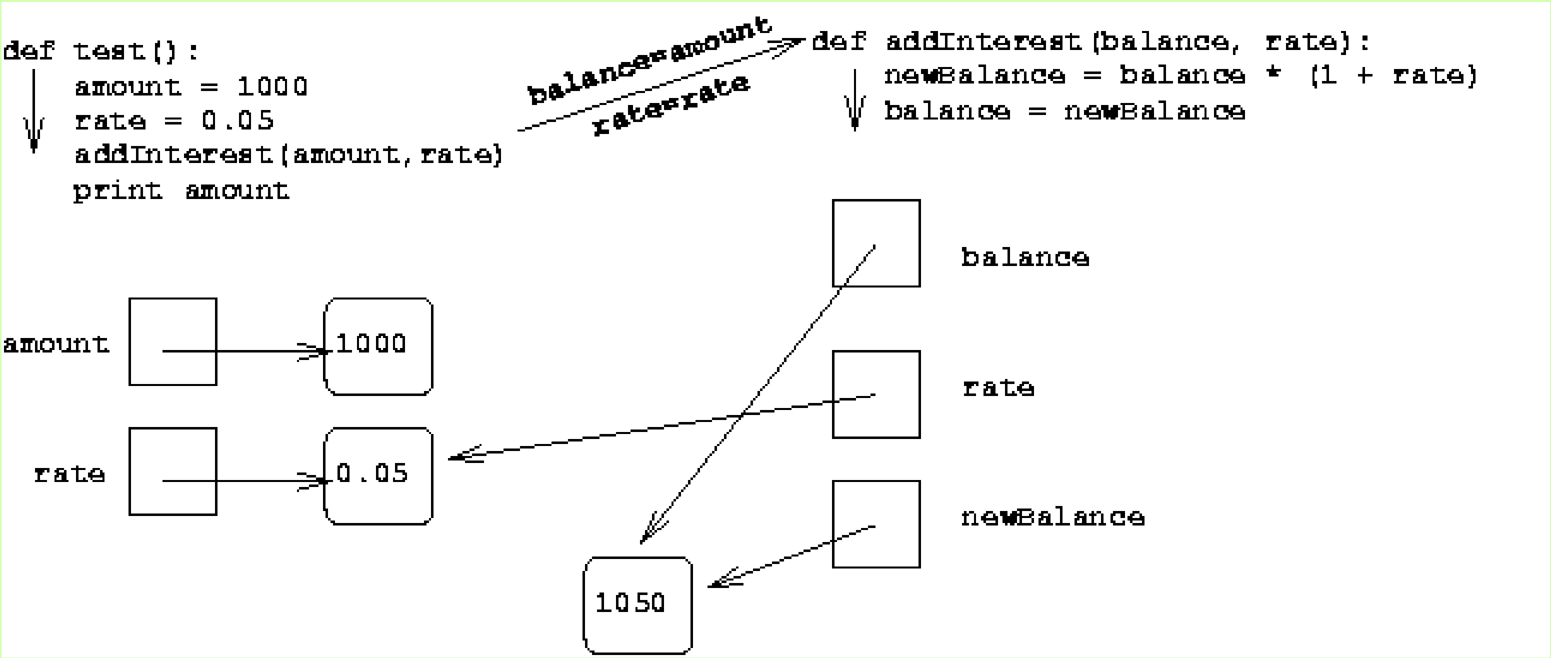
```
def addInterest(balance, rate):
    newBalance = balance * (1 + rate)
    balance = newBalance

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

Functions that Modify Parameters



Functions that Modify Parameters



Functions that Modify Parameters

- Some programming languages (C++, Ada, and many more) do allow variables themselves to be sent as parameters to a function. This mechanism is said to pass parameters *by reference*.
- When a new value is assigned to the formal parameter, the value of the variable in the calling program actually changes.

Functions that Modify Parameters

- Instead of looking at a single account, say we are writing a program for a bank that deals with many accounts. We could store the account balances in a list, then add the accrued interest to each of the balances in the list.
- We could update the first balance in the list with code like:

```
balances[0] = balances[0] * (1 + rate)
```

Functions that Modify Parameters

- This code says, “multiply the value in the 0^{th} position of the list by (1 + rate) and store the result back into the 0^{th} position of the list.”
- A more general way to do this would be with a loop that goes through positions 0, 1, …, length – 1.

Functions that Modify Parameters

```
# 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()
```

Functions that Modify Parameters

- Remember, our original code had these values:
  `[1000, 2200, 800, 360]`
- The program returns:
  `[1050.0, 2310.0, 840.0, 378.0]`
- What happened?
- It looks like `amounts` has been changed!

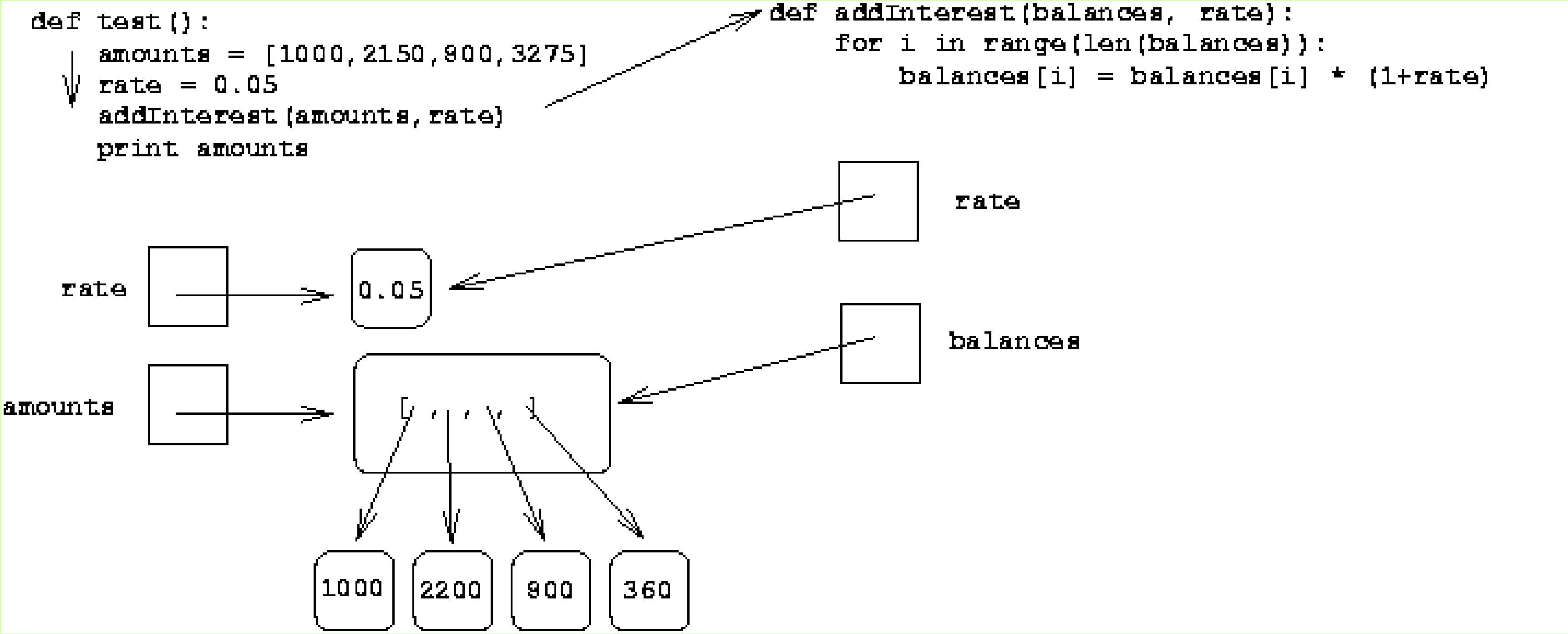
Functions that Modify Parameters

- The first two lines of `test` create the variables `amounts` and `rate`.
- The value of the variable `amounts` is a list object that contains four int values.

```
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
```

Functions that Modify Parameters



Functions that Modify Parameters

- 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
```

Functions that Modify Parameters

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

Piecewise Functions Example

$$f(n) = \begin{cases} 1 & \text{if } n = 1 \\ n - 1 & \text{if } n > 1 \end{cases}$$

f(4)

4 - 1

3

In Python

```
def f(n):
    if n == 1:
        return 1
    else:
        return n - 1
```

Fancier Functions

```
def f(n):
    return n + (n - 1)
```

Find f(4)

Fancier Functions

```
def f(n):
    return n + (n - 1)

def g(n):
    return n + f(n - 1)
```

Find g(4)

Fancier Functions

```
def f(n):
    return n + (n - 1)

def g(n):
    return n + f(n - 1)

def h(n):
    return n + h(n - 1)
```

Find h(4)

Recursion

```
def h(n):
    return n + h(n - 1)
```

- *h* is a *recursive* function,
  because it is defined in terms of itself.

Recursion

```
def h(n):
    return n + h(n - 1)
```

h(4)
4 + h(3)
4 + 3 + h(2)
4 + 3 + 2 + h(1)
4 + 3 + 2 + 1 + h(0)
4 + 3 + 2 + 1 + 0 + h(-1)
4 + 3 + 2 + 1 + 0 + -1 + h(-2)
...
Evaluating h leads to an infinite loop!

Recursion

```
def f(n):
    if n == 1:
        return 1
    else:
        return n+f(n - 1)
```

Find f(1)

Find f(2)

Find f(3)

Find f(100)

Recursion

```
def f(n):
    if n == 1:
        return 1
    else:
        return f(n - 1)

f(3)
f(3 - 1)
f(2)
f(2 - 1)
f(1)
1
```

Terminology

```
def f(n):
    if n == 1:
        return 1
    else:
        return n+f(n - 1)
```

base case

recursive case

"Useful" recursive functions have:

- at least one *recursive case*
- at least one *base case*
  so that the computation terminates

Recursion

```
def f(n):
    if n == 1:
        return 1
    else:
        return f(n + 1)
```

Find f(5)

We have a base case and a recursive case. What's wrong?

Recursion

The recursive case
should call the function
on a *simpler input*,
bringing us closer and closer
to the base case.

Factorial

- $4! = 4 \times 3 \times 2 \times 1 = 24$

Factorial

- $9! = 9 \times 8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1$
- $10! = 10 \times 9 \times 8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1$
- $10! = 10 \times 9!$
- $n! = n \times (n - 1)!$
- That's a recursive definition!

Factorial

```
def fact(n):
    return n * fact(n - 1)
```

fact(3)

$3 \times$ fact(2)

$3 \times 2 \times$ fact(1)

$3 \times 2 \times 1 \times$ fact(0)

$3 \times 2 \times 1 \times 0 \times$ fact(-1)

...

Factorial

- What did we do wrong?

- What is the base case for factorial?

Factorial

```
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n - 1)
```

fact(3)
$3 \times$ fact(2)
$3 \times 2 \times$ fact(1)
$3 \times 2 \times 1 \times$ fact(0)
$3 \times 2 \times 1 \times 1$
6

Fibonacci

In the Fibonacci sequence,
each term = sum of previous 2 terms

Let fib(n) be the n^{th} term.

Then, $\text{fib}(n) = \text{fib}(n - 1) + \text{fib}(n - 2)$

The sequence is defined recursively!

Fibonacci

```
def fib(n):
    return fib(n - 1) + fib(n - 2)
```

Find fib(1)

We need a base case!

Fibonacci

```
def fib(n):
    if n == 1:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)
```

Find fib(1)

Find fib(2)

How do we fix our function?

Fibonacci

```
def fib(n):
    if n <= 2:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)
```

Find fib(1)

Find fib(2)

Find fib(3)

Functions and Program Structure

- So far, functions have been used as a mechanism for reducing code duplication.
- Another reason to use functions is to make your programs more *modular*.
- As the algorithms you design get increasingly complex, it gets more and more difficult to make sense out of the programs.

Functions and Program Structure

- One way to deal with this complexity is to break an algorithm down into smaller subprograms, each of which makes sense on its own.