CMSC201 Computer Science I for Majors

Lecture 14 – Functions

Prof. Jeremy Dixon

Last Class We Covered

- Functions
  - Why they're useful
  - When you should use them
- Calling functions
- Variable scope
- Passing parameters

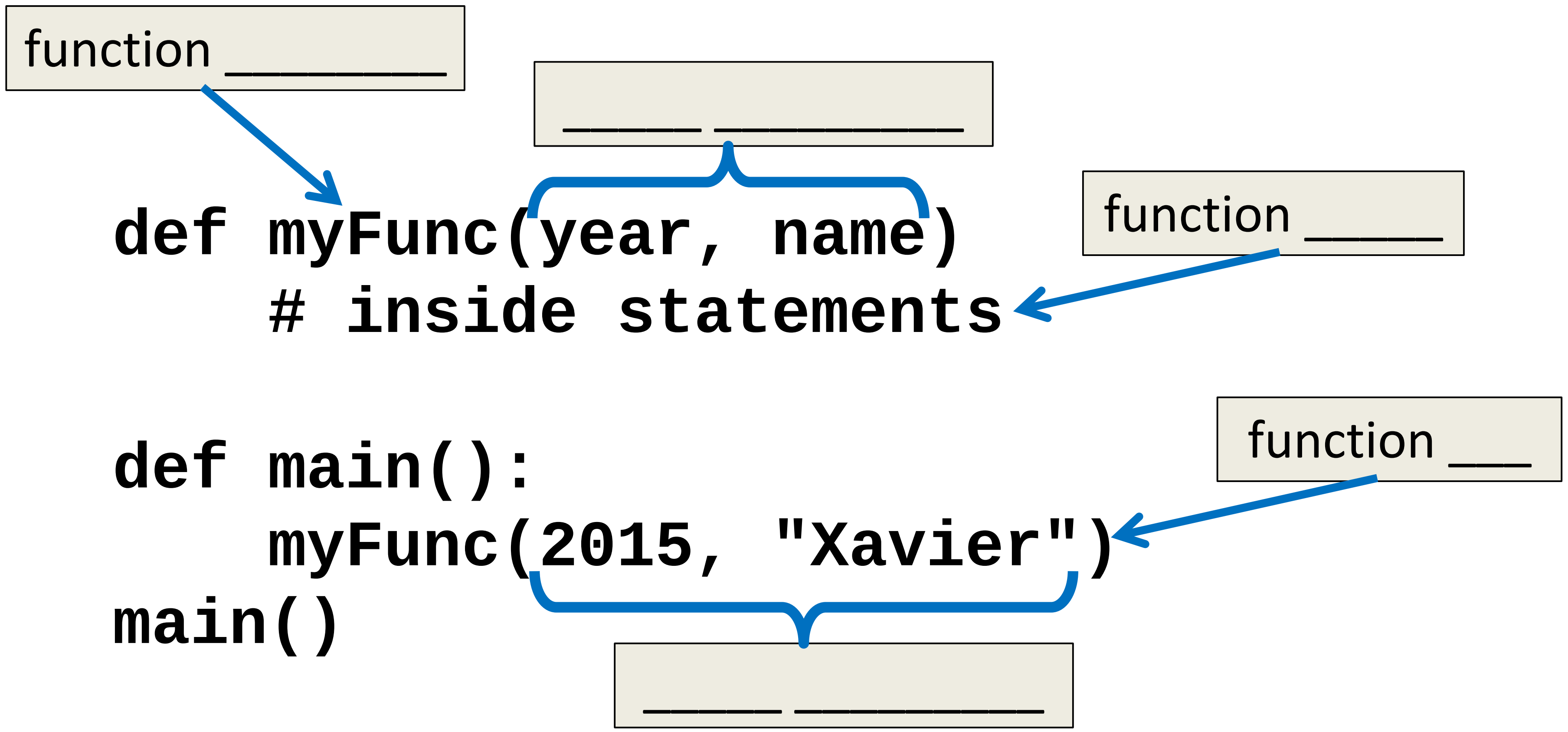
Any Questions from Last Time?

Today's Objectives

- To introduce value-returning functions (return)
- To understand how modifying parameters can change their values
- To practice function calls and some special situations
- To reinforce the value of modular programming

Function Review

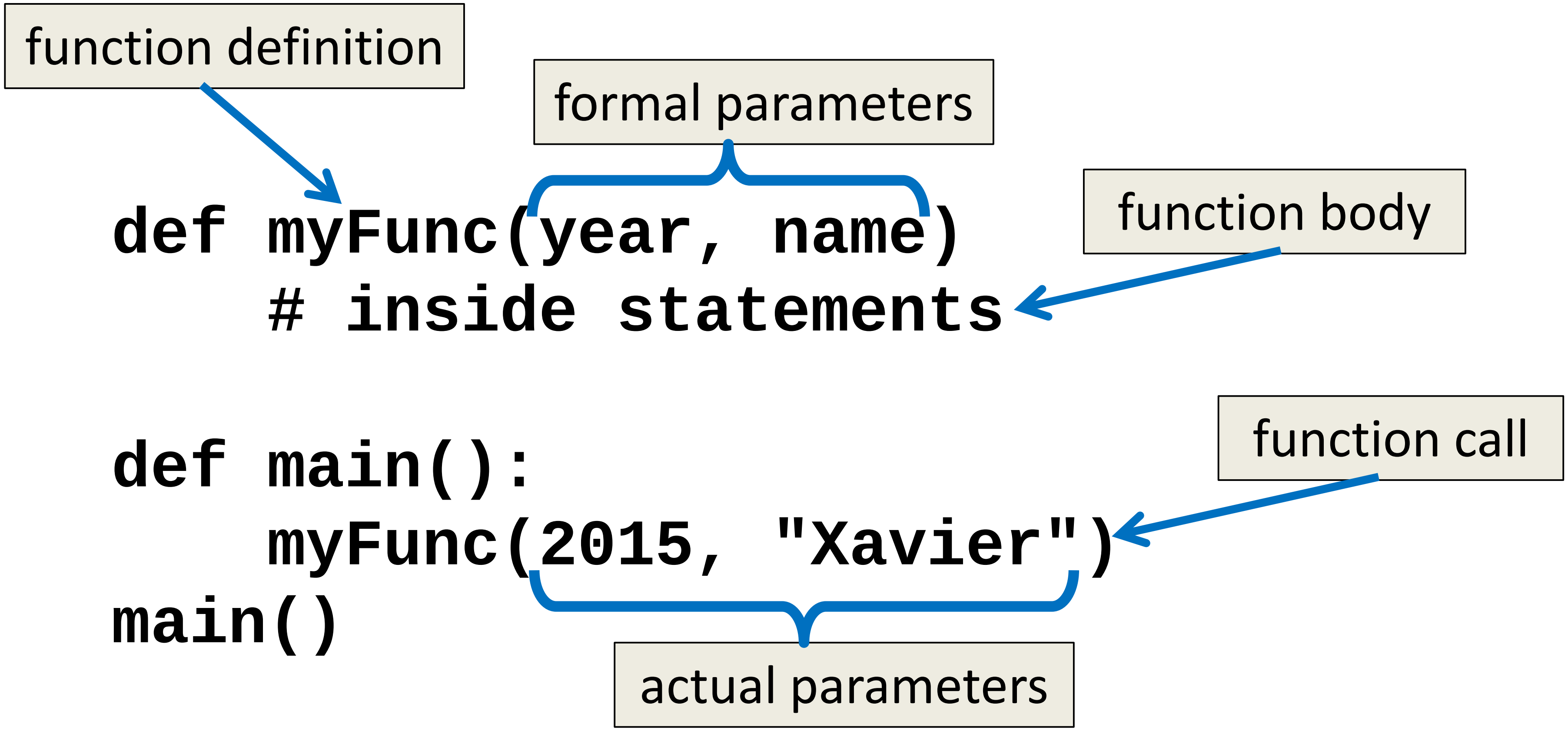
Function Vocabulary



```
def myFunc(year, name)
      # inside statements

def main():
      myFunc(2015, "Xavier")
main()
```

Function Vocabulary



```
def myFunc(year, name)
    # inside statements

def main():
    myFunc(2015, "Xavier")
main()
```

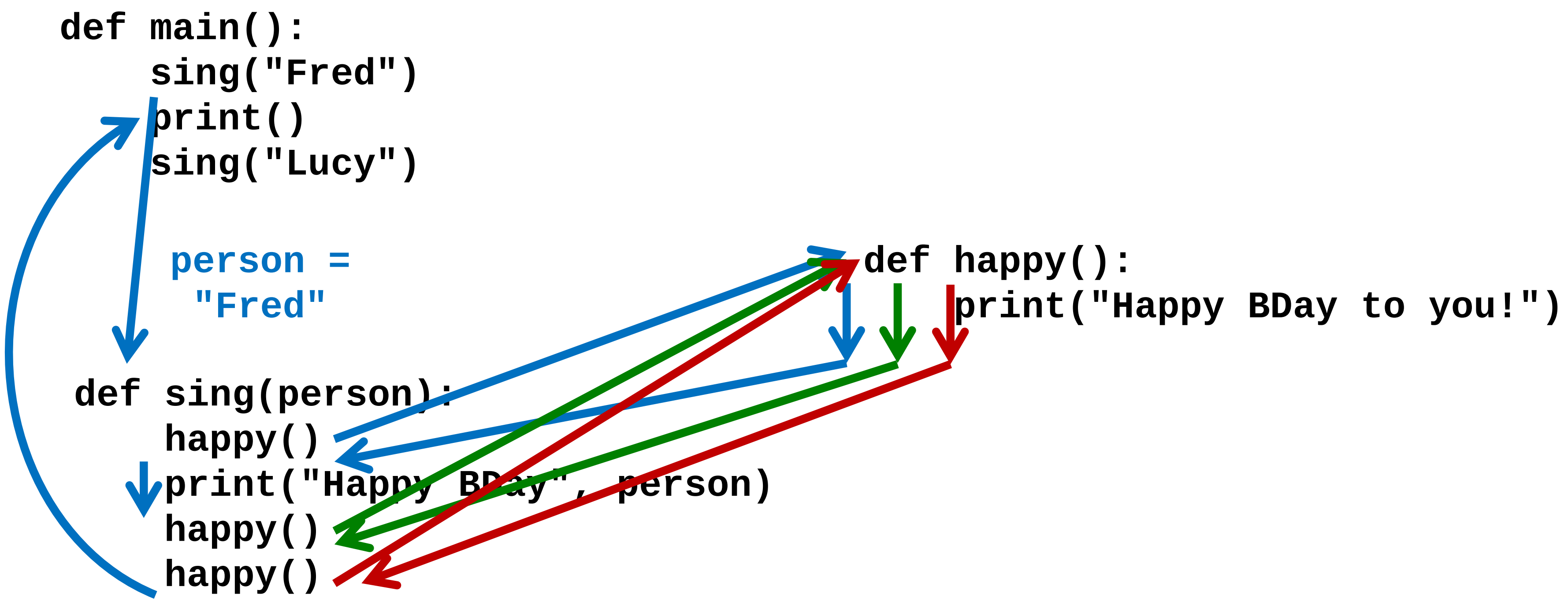
Visual Code Trace

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

```
def happy():
    print("Happy BDay to you!")
```

```
def sing(person):
    happy()
    print("Happy BDay", person)
    happy()
    happy()
```

Visual Code Trace



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

```
def sing(person):
    happy()
    print("Happy BDay", person)
    happy()
    happy()
```

```
def happy():
    print("Happy BDay to you!")
```

Note that the **`person`** variable in **`sing()`** disappeared!

Return Statements

Giving Information 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

Getting Information from a Function

- We've already seen numerous examples of functions that return values

    `int()` , **`str()`**, **`open()`**, **`input()`**, etc

- For example, **`int()`** takes in a string or double, and returns the integer of that
  - Or 0 if nothing is passed in to it

Functions that Return Values

- To have a function return a value after it is called, we need to use the **`return`** keyword

```
def square(num)
    # return the square
    return (num*num)
```

Handling Return Values

- When Python encounters **`return`**, it
  - Exits the function
  - Returns control back to where the function was called

- The value provided in the return statement is sent back to the caller as an expression result

Code Trace: Return from `square()`

Let's follow the flow of the code

```
def main():
    x = 5
    y = square(x)
    print(y)
main()
```

```
def square(num1):
    return num1 * num1
```

Step 1: Call **`main()`**
Step 2: Pass control to **`def main()`**
Step 3: Set **`x = 5`**
Step 4: See the function call to **`square()`**
Step 5: Pass control from **`main()`** to **`square()`**
Step 6: Set the value of **`num1`** in **`square()`** to **`x`**
Step 7: Return to **`main()`** and set **`y =`** return statement
Step 8: Print value of **`y`**

Code Trace: Return from **`square()`**

Let's follow the flow of the code

```
def main():
    x = 5
    y = square(x)
    print(y)
main()
```

```
def square(num1):
    return num1 * num1
```

Step 1: Call **`main()`**
Step 2: Pass control to **`def main()`**
Step 3: Set **`x = 5`**
Step 4: See the function call to **`square()`**
Step 5: Pass control from **`main()`** to **`square()`**
Step 6: Set the value of **`num1`** in **`square()`** to **`x`**
Step 7: Return to **`main()`** and set **`y =`** return statement
Step 8: Print value of **`y`**

Testing: Return from **`square()`**

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

Function with Multiple Return Values

Returning Multiple 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
```

Accepting Multiple Values

- When calling a function with multiple returns, use multiple assignments
- Assignment is based on position, just like passing in parameters is based on position

```
s, d = sumDiff(num1, num2)
```

Accepting Multiple Values

```
def main():
    num1 = int(input("Enter first number:  "))
    num2 = int(input("Enter second number: "))
    s, d = sumDiff(num1, num2)
    print("The sum is", s,
          "and the difference is", d)

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

main()
```

Accepting Multiple Values

```
def main():
    num1 = int(input("Enter first number:  "))
    num2 = int(input("Enter second number: "))
    s, d = sumDiff(num1, num2)
    print("The sum is", s,
          "and the difference is", d)

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

main()
```

s gets the first value returned

d gets the second value returned

Every Function Returns *Something*

- All Python functions return a value, whether they contain a **`return`** statement or not
- Functions without an explicit **`return`** hand back a special object, denoted **`None`**

Common Errors and Problems

- A common problem is writing a function that is expected to return a value, but forgetting to include the **`return`** statement
- If your value-returning functions produce strange messages, check to make sure you remembered to include the **`return`**!

Modifying Parameters

Other Ways to Pass Back Information

- Return values are the main way to send information back from a function
- We may also be able to pass information back by making changes directly to the parameters
- One of the problems with modifying parameters is due to the “scope” we discussed

Functions that Modify Parameters

- Suppose you are writing a program that manages bank accounts.
- One function we would need to create is one to accumulate interest on the account.

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

UMBC

AN HONORS UNIVERSITY IN MARYLAND

Functions that Modify Parameters

- The intent is to set the balance of the account to a new value that includes the interest amount.

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

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

Output

```
bash-4.1$ python interest.py
1000
bash-4.1$
```

Is this the expected output?

Functions that Modify Parameters

- We hope that that the 5% will be added to the amount, returning $1050
- Was $1000 the expected output?
- No – so what went wrong?
  - Let's trace through the program and find out

Functions that Modify Parameters

- First, we create two variables that are local to **`main()`**

Local Variables of **`main()`**

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

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

Functions that Modify Parameters

- Second, we call **`addInterest()`** and pass the local variables of **`main()`** as actual parameters

Call to **`addInterest()`**

Passing amount and rate, which are local variables

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

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

Functions that Modify Parameters

- Third, when control is passed to **`addInterest()`**, the formal parameters of (balance and rate) are set to the actual parameters of (amount and rate)

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

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

```
balance = amount
   rate = rate
```

Control passes to **`addInterest()`**

Functions that Modify Parameters

- Even though the parameter **`rate`** appears in both **`main()`** and **`addInterest()`**, they are separate because of scope

Even though rate is in both **`main()`** and **`addInterest()`**, they are in different places in memory

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

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

Functions that Modify Parameters

- In other words, 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*
- We call this passing parameters *by value*

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 called passing *by reference*

- When passing by reference, the value of the variable in the *calling program* actually changes

Functions that Modify Parameters

- Since Python doesn't have this capability, one alternative would be to change the **`addInterest`** function so that it returns the **`newBalance`**

Functions that Modify Parameters

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

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

test()
```

Code Trace (return statement)

Let's follow the flow of the code

```
def addInt(balance, rate):
    newBal = balance * (1 + rate)
    return newBal
```

```
def main():
    amount = 1000
    rate = 0.05
    amount = addInt(amount, rate)
    print(amount)
main()
```

Once we leave **addInt()**, the values of balance and rate are removed from memory

Step 1: Call **main()**
Step 2: Pass control to **def main()**
Step 3: Set **amount = 1000** and **rate = 0.05**
Step 4: Set amount = return statement of **addInt()**
Step 5: Pass control from **main()** to **addInt()**
Step 6: Set the value of **balance** in **addInt()** to amount
Step 7: Set the value of rate in **addInt()** to rate
Step 8: Set value of **newBal** to **balance * (1 + rate)**
Step 9: Return to **main()** and set value of **amount = newBal**
Step 10: Print value of **amount**

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 0th position of the list by (1 + rate) and store the result back into the 0th 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, rate)
    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? Python passes parameters by value, but 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)
```

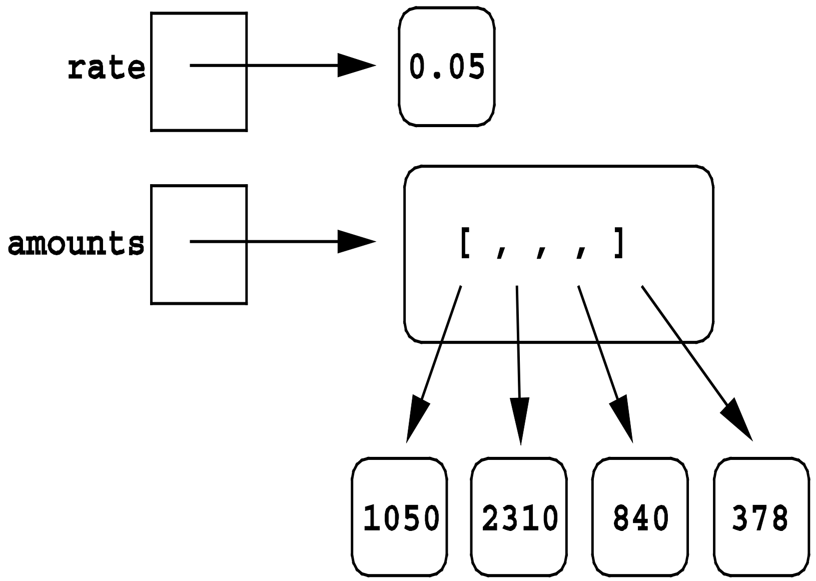
UMBC

AN HONORS UNIVERSITY IN MARYLAND

Functions that Modify Parameters

```
def test():
    amounts = [1000,2150,800,3275]
    rate = 0.05
    addInterest(amounts,rate)
    print amounts
```

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



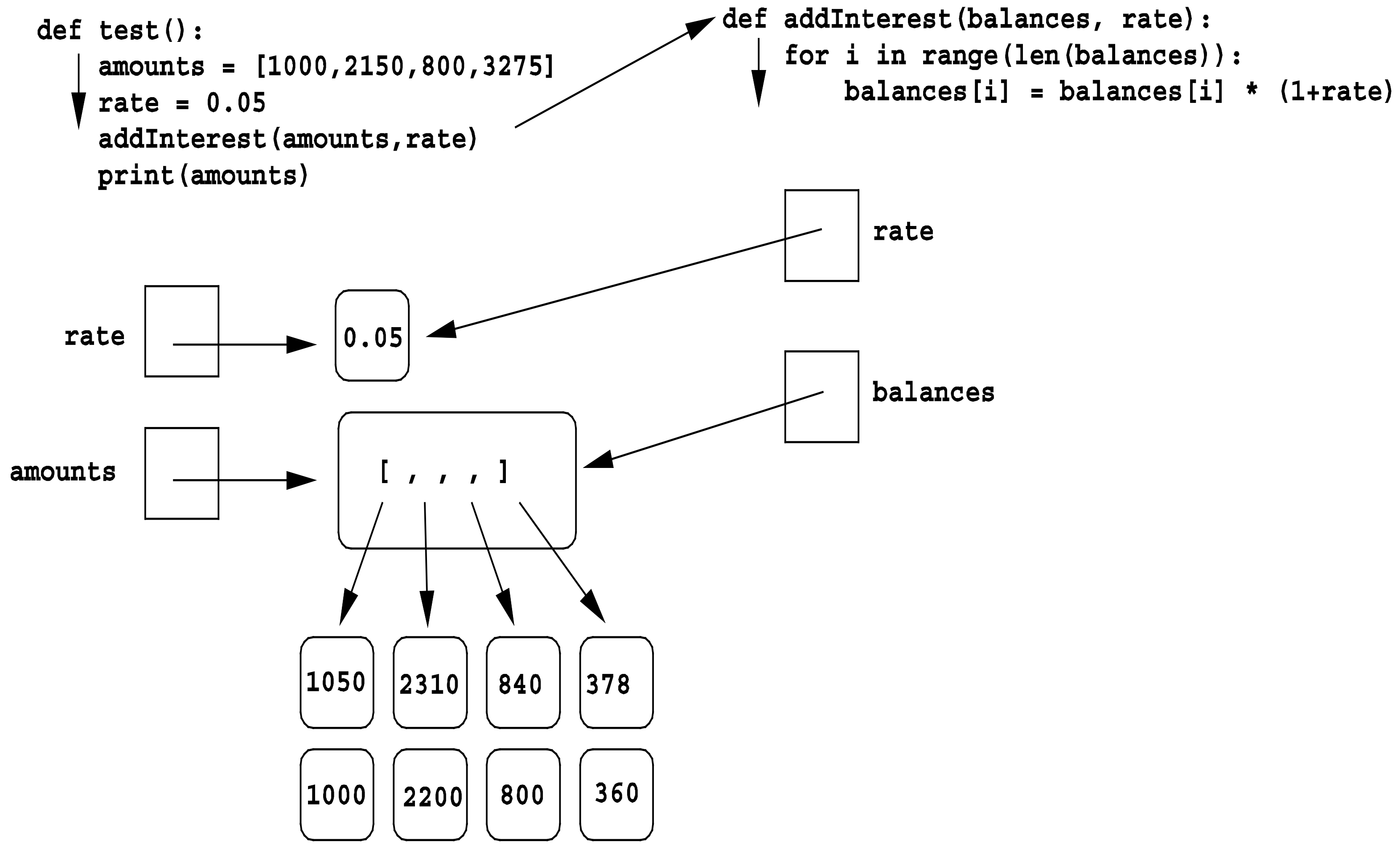
Functions that Modify Parameters

- 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



```
def test():
    amounts = [1000,2150,800,3275]
    rate = 0.05
    addInterest(amounts,rate)
    print(amounts)
```

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

Functions that Modify Parameters

- 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
```

Functions that Modify Parameters

- 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.

Function Call Exercise

Function Calls

- As we have previously discussed, function calls pass actual parameters to a function definition
- The question is: what are valid actual parameters in Python?

Valid or Invalid Function Calls

1. name = backwards(name)
2. intAge = 3calc(dob)
3. totalPay = totalCalc(rate, hours))
4. maxNum = avgScore(listOfScores, len(listOfScores))

1. Yes
2. No, invalid function name
3. No, extra parens
4. Yes, we can make function calls as an actual parameter.

Scope and Parameters

Mutable and Immutable

- In python, certain structures cannot change once they are created and they are called *immutable*.
  - They include integers, strings, and tuples
- Other structures can be changed after they are created and they are called *mutable*
  - They include lists, dicts, and user-defined lists

Scope in Functions

Pretend we call a function.
Depending on what type of data structure we are using, the data may permanently change

when a function is called, formal parameter B is assigned the actual parameter A

A is immutable
(int, string, tuple)

A **doesn't change**
If B changes

A is mutable
(lists, dicts, or user-defined)

B is assigned to something else
(B = "Hello")

A **doesn't change**
If B changes

B is modified in-place
(B.append(2))

A **changes**
If B changes

Functions that Modify Parameters

- Compared to other programming language such as C++, C, and Java, Python appears to always pass parameters *by value*

- However, as previously stated, mutable structures (lists, dicts, or user-defined) changes to the state of the object *will* be visible to the calling program

Modularity

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.

Any Other Questions?

Announcements

- We'll go over the exam in class on October 28th and 29th
- Homework 6 is out
  - Due by Thursday (Oct 22nd) at 8:59:59 PM
- Homework 7 will be out Oct 22nd
- Project 1 will be out Oct 29th