THE UNIVERSITY OF WESTERN AUSTRALIA

Lecture 11
Functions Part 2

Objectives

- To round out the discussion of functions.
- To learn how functions can change parameters.

Revision on Functions

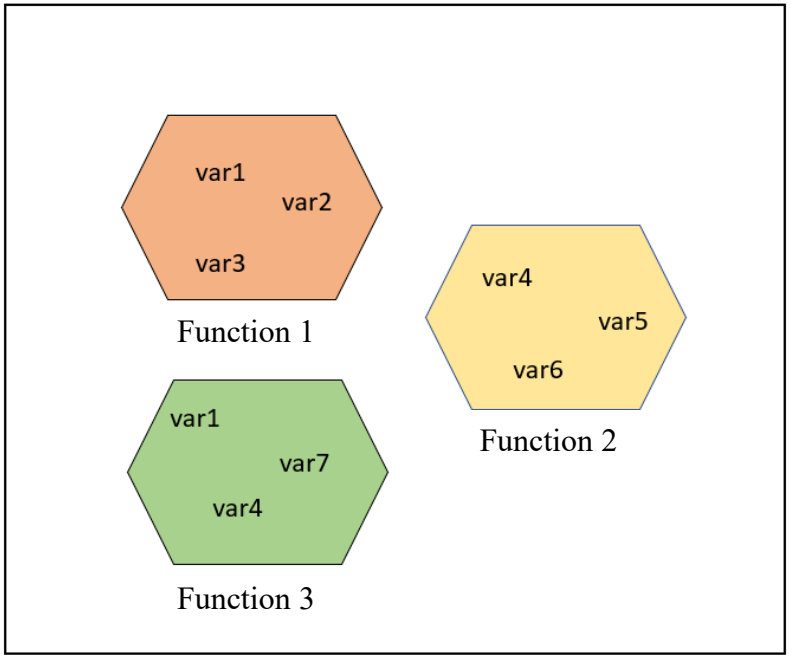
- Some of our programs comprised a single function called `main()`.
- We have already used built-in Python functions e.g. `abs()`, `int()`, `input()`, `print()` etc.
- We have used functions from the standard libraries e.g. `math.sqrt()`
- You may have defined your own functions during the labs, and Project, for answering different questions

Revision: Why use Functions?

- Having similar code in more than one place has some drawbacks.
  - *Having to type the same code twice or more.*
  - *Unnecessarily complicate the code*
  - *This same code must be maintained in multiple places. Will differ over time as code maintained*
- Functions are used to:
  - *avoid/reduce code duplication*
  - *make programs easy to understand*
  - *make programs easy to maintain.*

Revision: Scope of a Variable

- The *scope* of a variable refers to the places in a program a given variable can be referenced.
- The variables used inside of a function are *local* to that function, even if they happen to have the same name as the 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.



Functions that Modify Parameters

- Return values are the main way to send information from a function back to the caller.
- However, in certain circumstances 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 and one of the functions accumulates interest on the account.

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

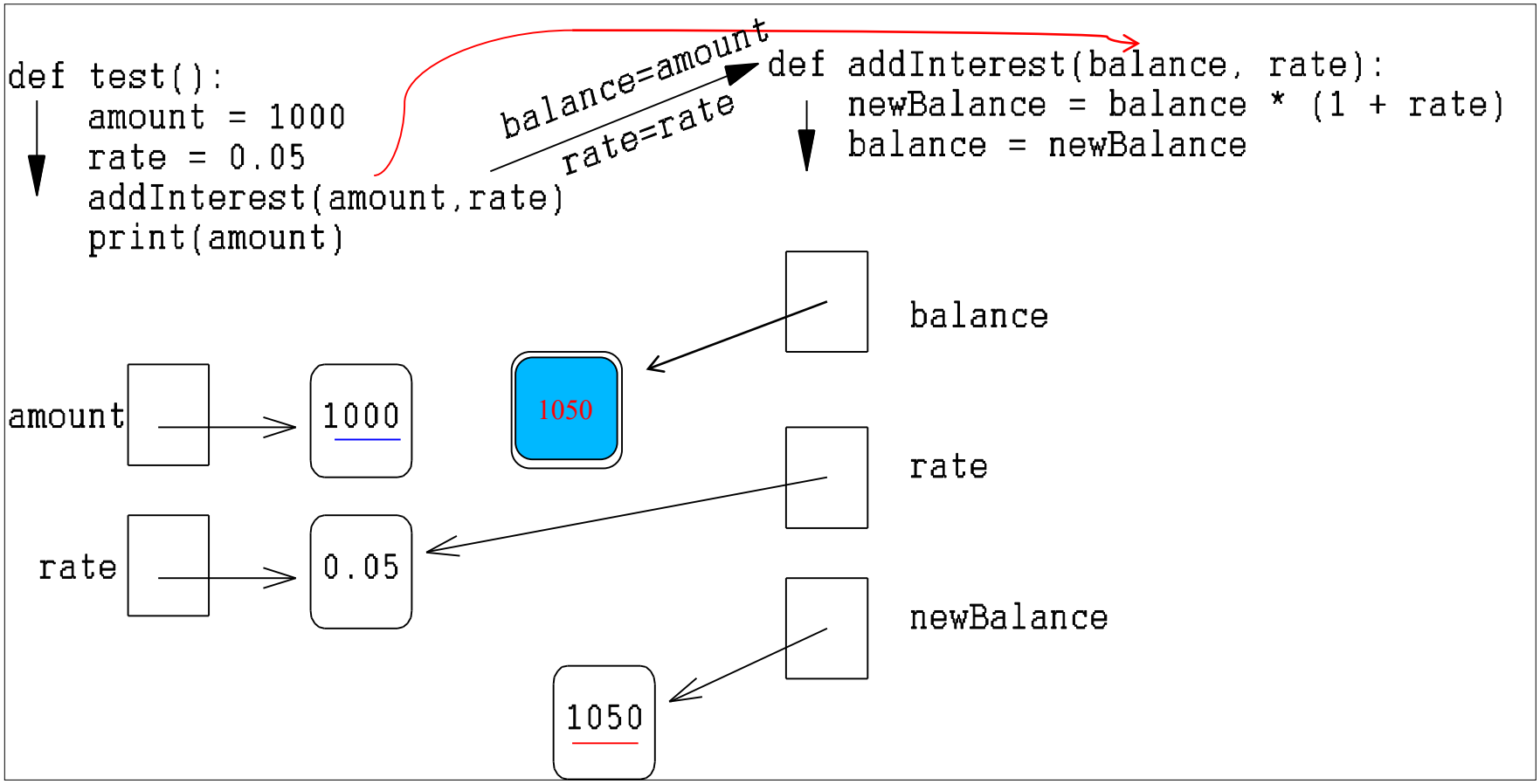
- Let's write a main program to test this:

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

```
>>> test()
1000
```

Is this a mistake? **NO**

Functions that Modify Parameters



The value of `amount` passed to `balance`. New value of `balance` computed, but not reflected back

Functions that Modify Parameters

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

Functions that Modify Parameters

Since Python doesn't pass arguments by-reference, one alternative is to change the `addInterest` function so that it returns `newBalance`.

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

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

>>> test()
1050
```

Functions that Modify Parameters

- Suppose we are writing a program 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)
```

- 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.

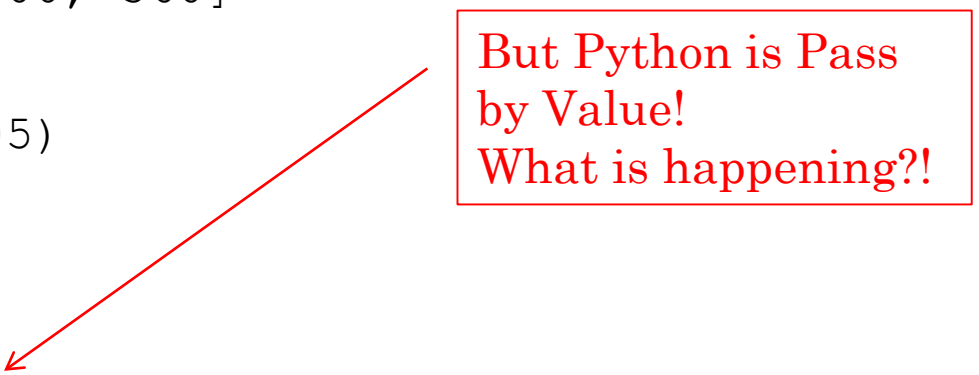
Functions that Modify Parameters

```
# 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()
[1050.0, 2310.0, 840.0, 378.0]
```



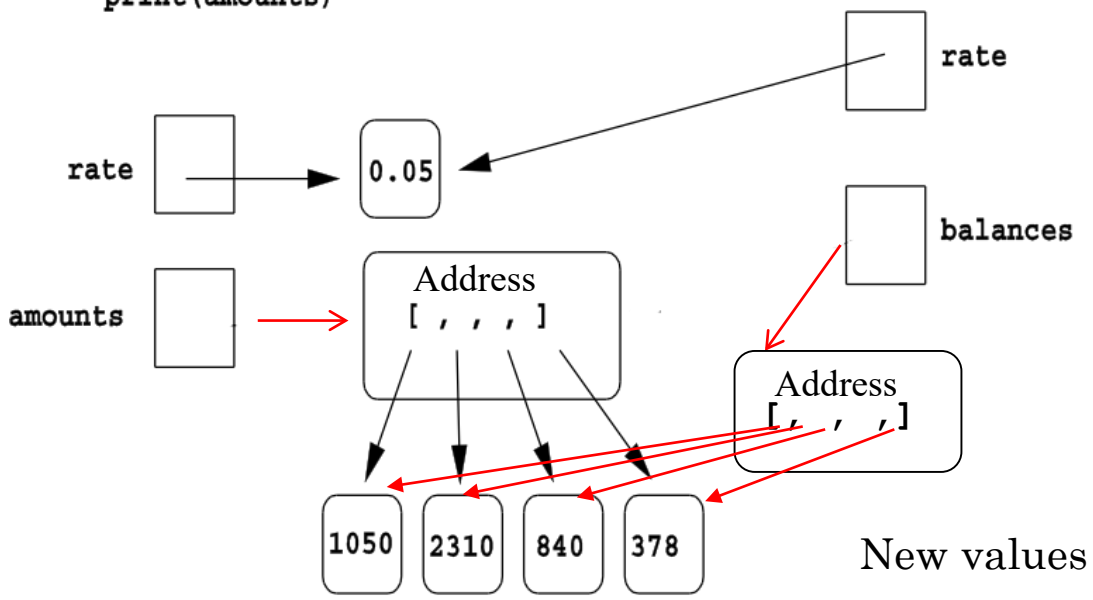
Functions that Modify Parameters

- When `addInterest` terminates, the list stored in `amounts` contains the new values.
- The variable `amounts` wasn't changed (it's the same list), but the contents of that list has changed, and this change is visible to the calling program.
- Parameters are always passed by value. However, if the value of the variable is a mutable object (like a list or other objects), then changes to the internal state of the object ***will*** be visible to the calling program.

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)
```



`amounts` and `balances` refer to the same object

Functions that Modify Parameters

Two names: Nanga Parbat, Killer Mountain (Himalayas, Pakistan)

https://en.wikipedia.org/wiki/Nanga_Parbat (9th highest mountain in world)

Two identifies for the same object – just the contents have been changed

Default Values for Parameters

- The most common way to call functions is to provide N values for N parameters.
- However, sometimes it's handy to be able to ignore less important parameters and just have default values.
- For example, you wish to define the function `mean()`, but offer a range of different interpretation of mean, e.g. arithmetic (i.e. standard), geometric mean and harmonic mean. The function definition could begin:

```
def mean(values, type="arithmetic") :
  if type == "arithmetic" :
     ............
  elif type == "geometric" :
     ..............
```

Default Values for Parameters

- `mean([1,2,3,4,5])` is the same as calling `mean([1,2,3,4,5], "arithmetic")`, but if you want the geometric mean, that has to be called explicitly, `mean([1,2,3,4,5], "geometric")`
- One Gotcha. The parameters with default values have to come **after** the positional parameters.
- Upside: Only the important parameters (which come first) need by specified

Functions and Program Structure

- So far, functions have been used as a mechanism for reducing code duplication.
- As the algorithms you design get increasingly complex, it gets more and more difficult to make sense out of the programs.
- One way to deal with this is to make your programs more modular.
  - *Recall problem decomposition*
  - *One way to deal with this complexity is to break an algorithm down into smaller subprograms, each of which makes sense on its own.*
  - *Separation of concerns*

Functions and Program Structure

- For example, a function at the start can deal with user input data
  - *Check that inputs are of the expected type and range.*
  - *Once the input data is checked and known to be sound, another function (set of functions) can process the data*

Summary

- We learned the advantages of dividing a program into multiple cooperating functions.
- We studied how functions can change parameters.