PYTHON: FUNCTIONS

Objectives

- To understand why programmers divide programs up into sets of cooperating functions.
- To be able to define new functions in Python.
- To understand the details of function calls and parameter passing in Python.
- To write programs that use functions to reduce code duplication and increase program modularity.
- To understand scope of variables.
- Become familiar with the use of docstrings in Python.

Functions everywhere

- So far, we've seen three different types of functions:
    - Our programs comprise a single function called `main()`.
    - Built-in Python functions (e.g., `print()`)
    - Functions from the standard libraries (e.g., `math.sqrt()`)
- Why functions?
    - Code reuse
    - Easier to maintain (modularity)
    - Facilitate solving a complex problem

Parameters = arguments

Function definition

- The part of the program that creates a function is called a *function definition*.

```
def <name>(<formal-parameters>):
     <body>
```

- E.g.,

```
def avg(n1, n2, n3):
   print((n1 + n2 + n3) / 3)
```

  - where `n1, n2, n3` are *formal parameters.*
- A function is called/invoked by `<name>(<actual-parameters>)`
  - E.g., `avg(1, 2, 3)`, where 1, 2 and 3 are *actual parameters*.
- Functions must be defined/imported before it is called.

(Non-)Value-returning functions

- The three arguments in `avg()` are passed to the three parameters according to their positions.
- A **non-value-returning function** is called not for a returned value, but for its *side effects*.
  - A **side effect** is an action other than returning a function value, such as displaying output on the screen.
  - There is no **`return`** statement.
- A **value-returning function** is a program routine called for its return value.
  - The *return statement* of the form **`return`** `expr`, where `expr` may be any expression.

Getting Results from a Function

- Passing parameters provides a mechanism for initializing the variables in a function.
- Parameters act as “inputs” to a function.
- The function may return the result through the `return` statement.
- E.g.,
  ```
  def square(x):
       return x * x
  ```
- When `return` is encountered in a function, the execution terminates immediately and get back to the caller of the function.

Returning multiple values

- In Python, it allows a function to return more than one value (in terms of a *tuple*)
- To do this, simply list more than one expression in the `return` statement.
- E.g.,

```
def sumDiff(x, y):
     sum = x + y
     diff = x - y
     return sum, diff
x, y = sumDiff(9, 8)
```

Positional and keyword arguments

- A **positional argument** is an argument that is assigned to a particular parameter based on its position in the argument list.
- A **keyword argument** is an argument that is specified by parameter name.

- F(parameter = **argument / value)**
- **Avg(n1= 1, n2=2,n3=3)**
- **Ave(1,2,3)**

EXERCISE 1

Try

```
def printNumbers(n1, n2, n3):
    print(n1, n2, n3)

m1, m2, m3 = 10, 20, 30
printNumbers(m1, m2, m3)
printNumbers(n2=m2, n1=m3, n3=m1)
```

Default parameters

- A **default argument** is an argument that can be optionally provided in a given function call.
- When not provided, the corresponding parameter provides a default value.
- For example, the followings give the same outputs.
  - `print("The default print() parameters.")`
  - `print("The default print() parameters.", end="\n")`
  - `print("The default print() parameters.", sep=" ")`
  - `print("The default print() parameters.", end="\n", sep=" ")`
  - `print("The default print() parameters.", sep=" ", end="\n")`

Functions that modify parameters

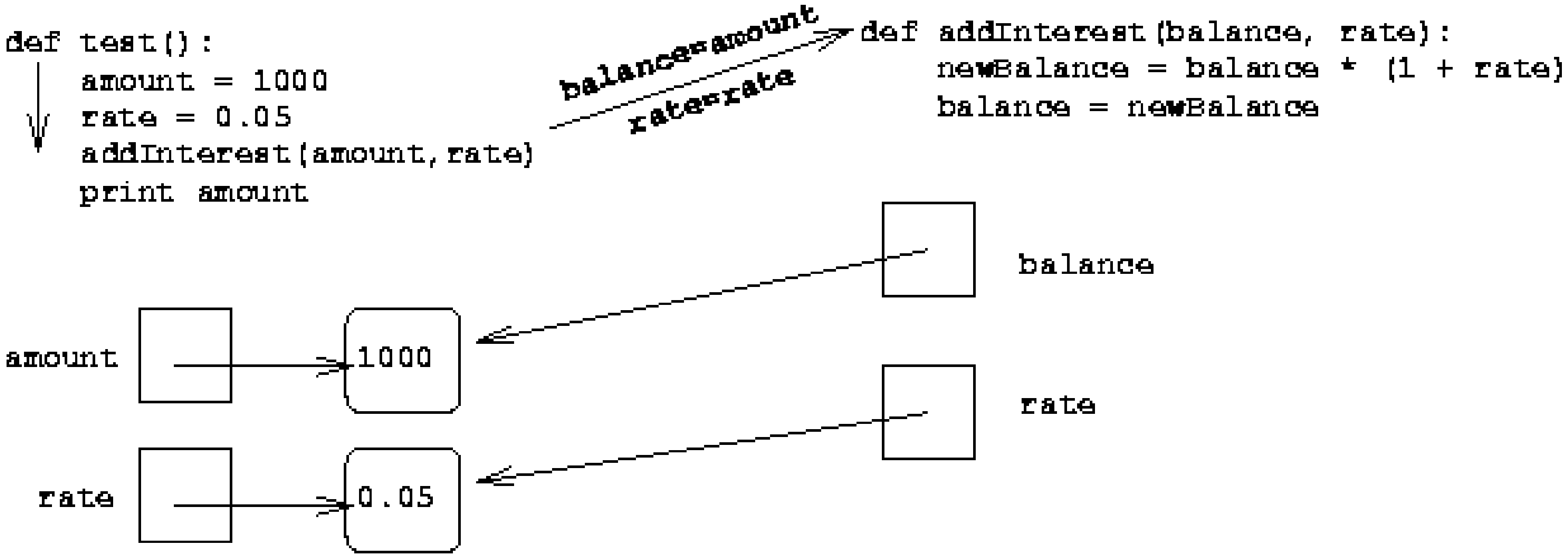
- Return values are the main way to send information from a function back to the caller.
- Can we communicate back to the caller by making changes to the function parameters?

EXERCISE 2

Try whether this works.

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

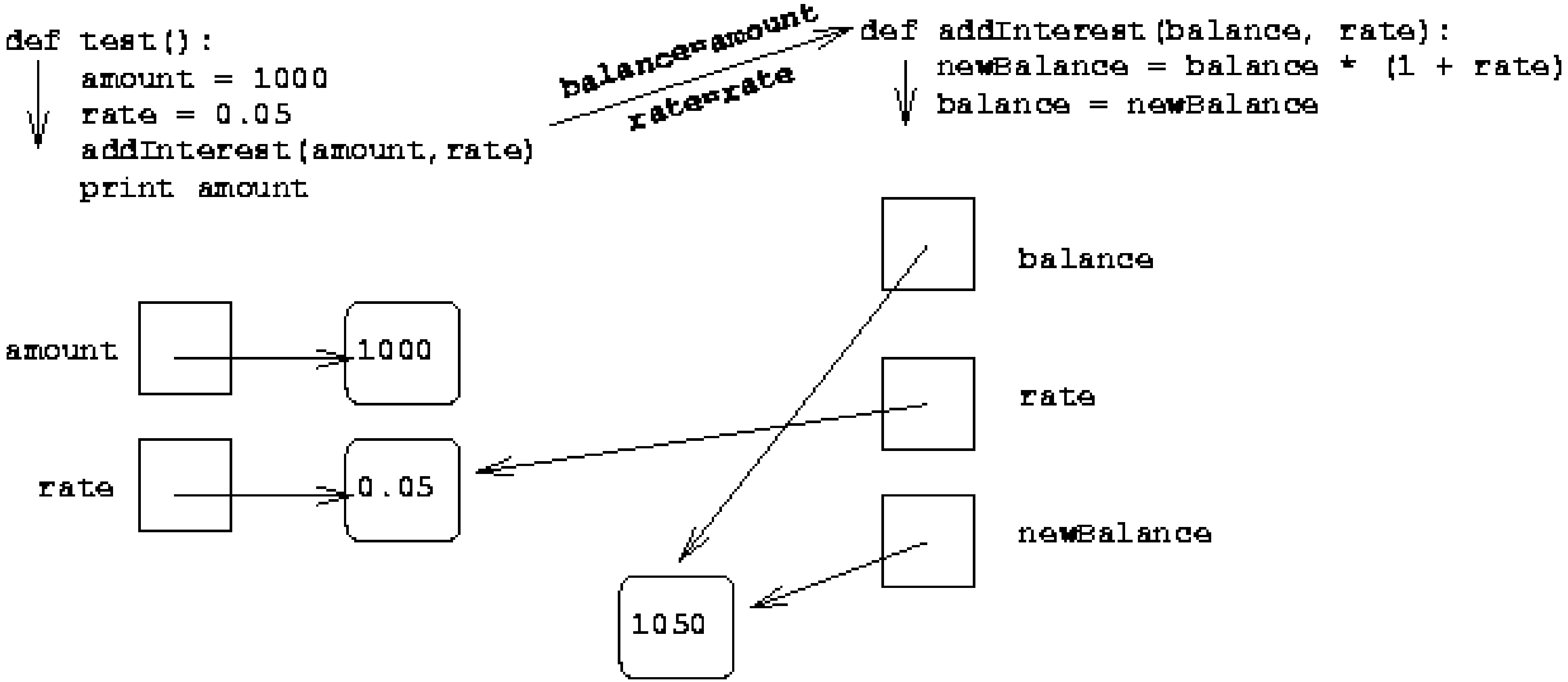
Behind the scene



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

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

Behind the scene



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

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

EXERCISE 3

Will this work?

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

def test():
    amount = 1000
    rate = 0.05
    addInterest(amount, rate)
    print("My current balance is: ", amount)
test()
```

At the end

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

Another example

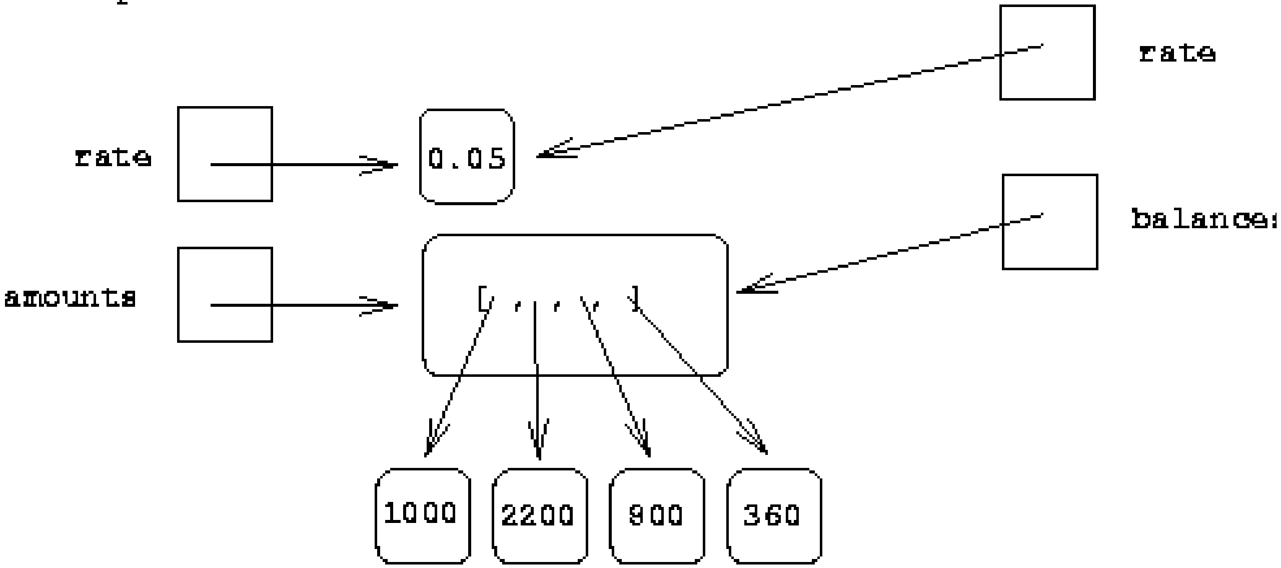
```
def addInterest(balances, rate):
    for i in range(len(balances)):
        balances[i] = balances[i] * (1+rate)

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

test()
```

Is the output expected?

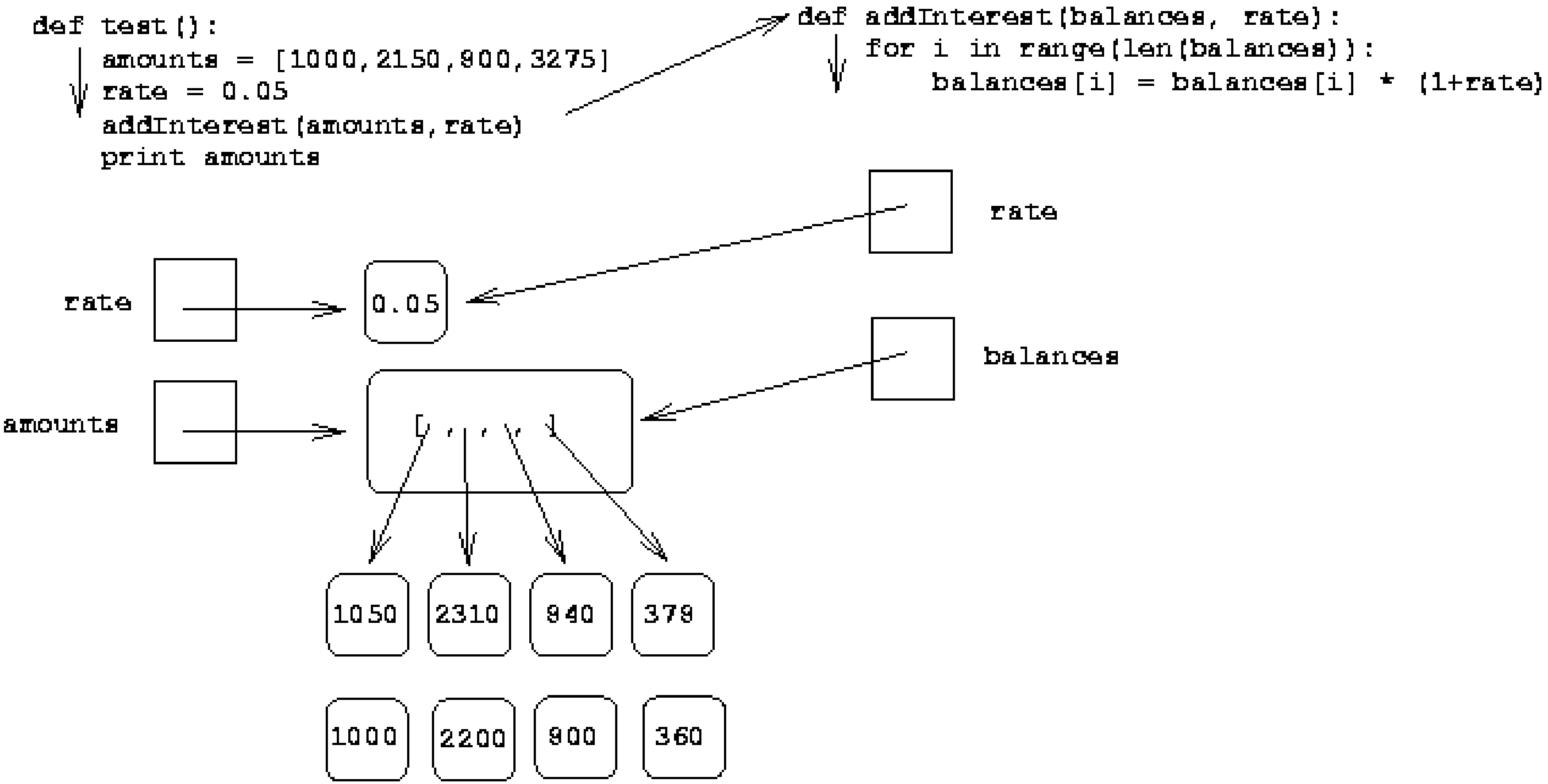
Behind the scene

```
def test():
    amounts = [1000,2150,900,3275]
    rate = 0.05
    addInterest(amounts,rate)
    print amounts
```

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



Behind the scene



```
def test():
    amounts = [1000,2150,900,3275]
    rate = 0.05
    addInterest(amounts, rate)
    print amounts
```

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

A few points

- Each value is given by multiple names (i.e., name aliasing)
- The old values have not changed, but the new values were created and assigned into the list.
- The old values will be destroyed during *garbage collection*.
- Will this work?

```
def changeString(s):
     s[0] = "A"

name = "DennisLiu"
changeString(name)
```

EXERCISE 4

Try

```
def funList(aList):
        aList.append(1)
        aList = [2,3]

L1 = [0]
funList(L1)
print(L1)
```

Draw a similar diagram as that in slide 18 to illustrate what behind the scene is.

Global and local variables

- A *local variable* is a variable that is only accessible from within the function it resides. Such variables are said to have *local scope*.
- A *global variable* is a variable defined outside of any function definition. Such variables are said to have *global scope*.
- The use of global variables is considered bad programming practice.

Exercise 5

- Try the code. What will be printed out?
- Explain what you see.
- Then, in `funA()`, add x = 3 at the end. What do you see? Why?

```
def funA():
     print(x)

def funB():
     # x is a local
variable
     x = 2
     print(x)

# x is a global variable
x = 10
funA()
funB()
print(x)
```

Scope resolution

- Scope of a name (identifier) is the set of program statements over which it can be referred to.
- The scope of a name depends on where it is created and whether the name is used somewhere else.
  - Local and global scopes
- When encountering a name, Python interpreter will search it in the order below:
  - Local
  - Enclosing (do not bother in this course)
  - Global
  - Built-in

EXERCISE 6

Run the code below and try to explain the result.

```
x = 1
def fun():
    x = x + 1
    print(x, "x inside fun()")
print(x, "x outside fun()")
fun()
```

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.

Where do the modules come from?

- The term “module” refers to the design and/or implementation of specific functionality to be incorporated into a program.
- Modular-II

```
MODULE Hello;
FROM STextIO IMPORT WriteString;
BEGIN
   WriteString("Hello World!");
END Hello.
```

- A Python module is a file containing Python definitions and statements.
  - By convention, modules are named using all lowercase letters and optional underscore characters.

EXERCISE 7

- Import a module. (e.g., `math`)
- Use `help(module_name)` or `dir(module_name)` to find out the functions available in that module.
- Use `print(function_name.__doc__)` to print out what the function does.

Interface, client and docstring

- A module’s *interface* is a specification of what it provides and how it is to be used.
- Any program code making use of a given module is called a *client* of the module.
- A *docstring* is a string literal denoted by triple quotes used in Python for providing the specification of certain program elements.

Documentation String (docstring)

(http://legacy.python.org/dev/peps/pep-0008/#documentation-strings)

- A *docstring* is a string literal denoted by triple quotes given as the first line of certain program elements.
- These additional lines must be indented at the same level.
- The docstring for a module is simply the first block of quoted text to appear in the module.
- Write docstrings for all public modules, functions, classes, and methods.
- Convention:

```
"""Return an integer which is the sum of a, b.
This is the second line.
This is the last line."""
```

EXERCISE 8

- Create a program with a `sum()` function that add **two** input parameters and return the sum.
- Create also a `main()` function to test the sum() function.
- Add *docstring* to both functions.
- How can you check if *docstring* is properly typed?

A Final Recap

- In Python, functions can be given any name (except for keywords).
- Although there is no specific requirement for the start function, it is always a good practice to call the start function **`main()`**.
  - Basically, most Python programmers will develop a function called **`main()`** and this function will be called whenever the Python program is executed.
- In C/C++/Java, the start function must always be called **`main()`** and when the program is executed, **`main()`** will be automatically executed.
  - That is the reason why you could often see **`main`**() being defined in our Python programs, and the first line of code (sometimes the only line of code) at the bottom is **`main()`**.

END