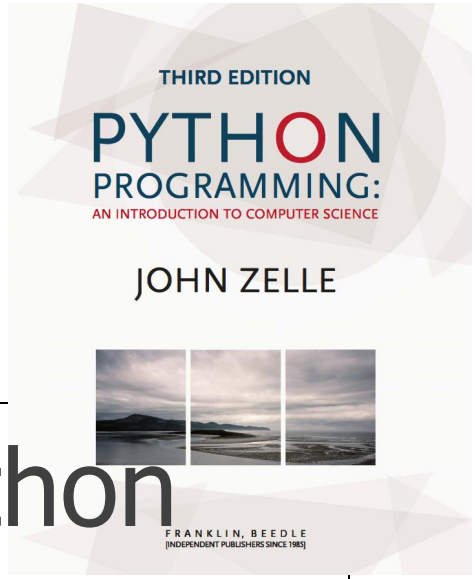


An Introduction to Python Programming

Chapter 6: Defining 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.

The Function of Functions

- So far, we've seen **four different** types of functions:
  - Our programs comprise a single function called main().
  - Built-in Python functions (abs)
  - Functions from the standard libraries (math.sqrt)
  - Functions from the graphics module (p.getX())

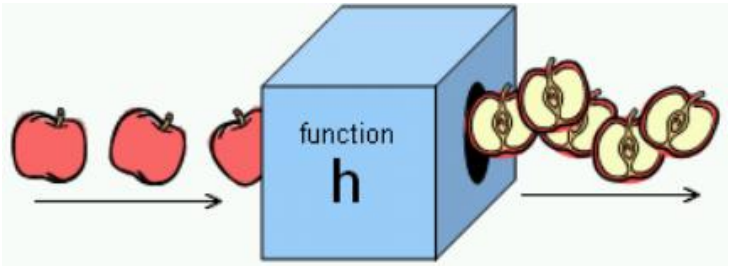
The Function of Functions

- In the futval.py, the code for drawing the bars occurs **in two different places**.
  - Issue one: writing the same code twice or more.
  - Issue two: This same code must be maintained in two separate places.

- Functions can be used to **reduce code duplication** and make programs more easily understood and maintained.

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**. We can then execute this sequence at any time by referring to the name.



Functions, Informally

- The part of the program that creates a function is called **a *function definition***.
- When the function is used in a program, we say the definition is ***called* or *invoked***.

Functions, Informally

- Happy Birthday lyrics…

```
>>> def main():
        print("Happy birthday to you!")
        print("Happy birthday to you!")
        print("Happy birthday, dear Fred.")
        print("Happy birthday to you!")
```

- Gives us this...

```
>>> main()
Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Fred.
Happy birthday to you!
```

Functions, Informally

- The **duplicated code :**

```
print("Happy birthday to you!")
```

- We can **define a function** to print out this line:

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

- With this function, we can rewrite our program.

Functions, Informally

- The new program

```
>>> def singFred():
        happy()
        happy()
        print("Happy birthday, dear Fred.")
        happy()
```

- Gives us this output

```
>>> singFred()
Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Fred.
Happy birthday to you!
```

Functions, Informally

- What if it's Lucy's birthday? We could write a new singLucy function!

```
>>> def singLucy():
        happy()
        happy()
        print("Happy birthday, dear Lucy.")
        happy()
```

Functions, Informally

- We could write **a main program** to sing to both Lucy and Fred

```
>>> def main():
        singFred()
        print()
        singLucy()
```

- this new output :

```
>>> main()
Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Fred.
Happy birthday to you!

Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Lucy.
Happy birthday to you!
```

Functions, Informally

- But… there's **still a lot of code duplication**.
- The only difference between ***singFred*** and ***singLucy*** is the name.These two routines could be collapsed together by using a ***parameter***.

```
>>> def sing(person):
        happy()
        happy()
        print("Happy Birthday, dear", person + ".")
        happy()
```

Functions, Informally

- The generic function ***sing***
- This function uses **a parameter named person**.
- A *paramater* is a **variable** that is **initialized** when the function is called.
- We can put together a new main program!

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

Future Value with a Function

- In the future value graphing program, we see similar code:

```
# Draw bar for initial principal
bar = Rectangle(Point(0, 0), Point(1, principal))
bar.setFill("green")
bar.setWidth(2)
bar.draw(win)

bar = Rectangle(Point(year, 0), Point(year+1, principal))
bar.setFill("green")
bar.setWidth(2)
bar.draw(win)
```

Future Value with a Function

- To properly draw the bars, we need **three pieces of information**.
  - The **year** the bar is for
  - How **tall** the bar should be
  - The **window** the bar will be drawn in
- These three values can be supplied as **parameters** to the function.

Future Value with a Function

- The resulting function looks like this:

```
def drawBar(window, year, height):
    # Draw a bar in window for given year with given height
    bar = Rectangle(Point(year, 0), Point(year+1, height))
    bar.setFill("green")
    bar.setWidth(2)
    bar.draw(window)
```

- If win is a Graphwin, we can draw a bar for year 0 and principal of $2000 using this call:

```
drawBar(win, 0, 2000)
```

Future Value with a Function

```
# futval_graph3.py
from graphics import *

def drawBar(window, year, height):
    # Draw a bar in window starting at year with given height
    bar = Rectangle(Point(year, 0), Point(year+1, height))
    bar.setFill("green")
    bar.setWidth(2)
    bar.draw(window)

def main():
    # Introduction
    print("This program plots the growth of a 10-year investment.")

    # Get principal and interest rate
    principal = float(input("Enter the initial principal: "))
    apr = float(input("Enter the annualized interest rate: "))

    # Create a graphics window with labels on left edge
    win = GraphWin("Investment Growth Chart", 320, 240)
    win.setBackground("white")
```

Future Value with a Function

```
    win.setCoords(-1.75,-200, 11.5, 10400)
    Text(Point(-1, 0), ' 0.0K').draw(win)
    Text(Point(-1, 2500), ' 2.5K').draw(win)
    Text(Point(-1, 5000), ' 5.0K').draw(win)
    Text(Point(-1, 7500), ' 7.5k').draw(win)
    Text(Point(-1, 10000), '10.0K').draw(win)

    drawBar(win, 0, principal)
    for year in range(1, 11):
        principal = principal * (1 + apr)
        drawBar(win, year, principal)

    input("Press <Enter> to quit.")
    win.close()
main()
```

Functions and Parameters: The Details

- But why is **window** also a parameter to this function?
- The ***scope*** of a variable refers to the places in a program a given variable can be referenced.
- Each function is its own little subprogram. The **variables** used inside of one function are ***local*** to that 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 and Parameters: The Details

- Since the **`GraphWin`** in the variable **`win`** is created inside of **`main`**, it is not directly accessible in **`drawBar`**.
- The **`window`** parameter in **`drawBar`** gets assigned the value of **`win`** from **`main`** when **`drawBar`** is called.

Functions and Parameters: The Details

- A **function definition** looks like this:

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

  <body>

  - The name of the function must be an **identifier**
  - Formal parameters, like all variables used in the function, are **only accessible** in the body of the function.
  - A function is called by using its name followed by a list of *actual parameters* or *arguments*. **<name>(<actual-parameters>)**

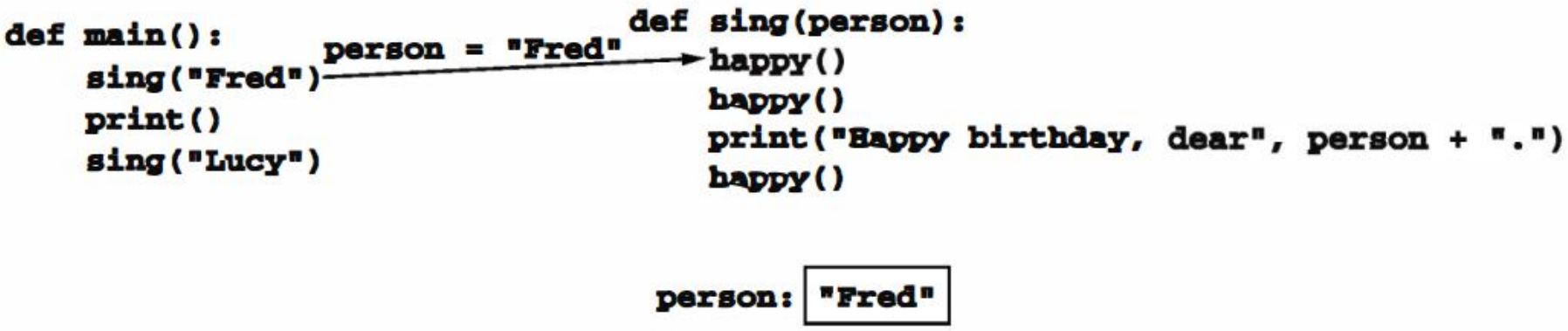
Functions and Parameters: The Details

- When Python comes to a function call, it initiates a four-step process.
  - The calling program **suspends** execution at the point of the call.
  - The formal parameters of the function **get assigned** the values supplied by the actual parameters in the call.
  - The body of the function is **executed**.
  - Control **returns** to the point just after where the function was called.

Functions and Parameters: The Details

- For example:

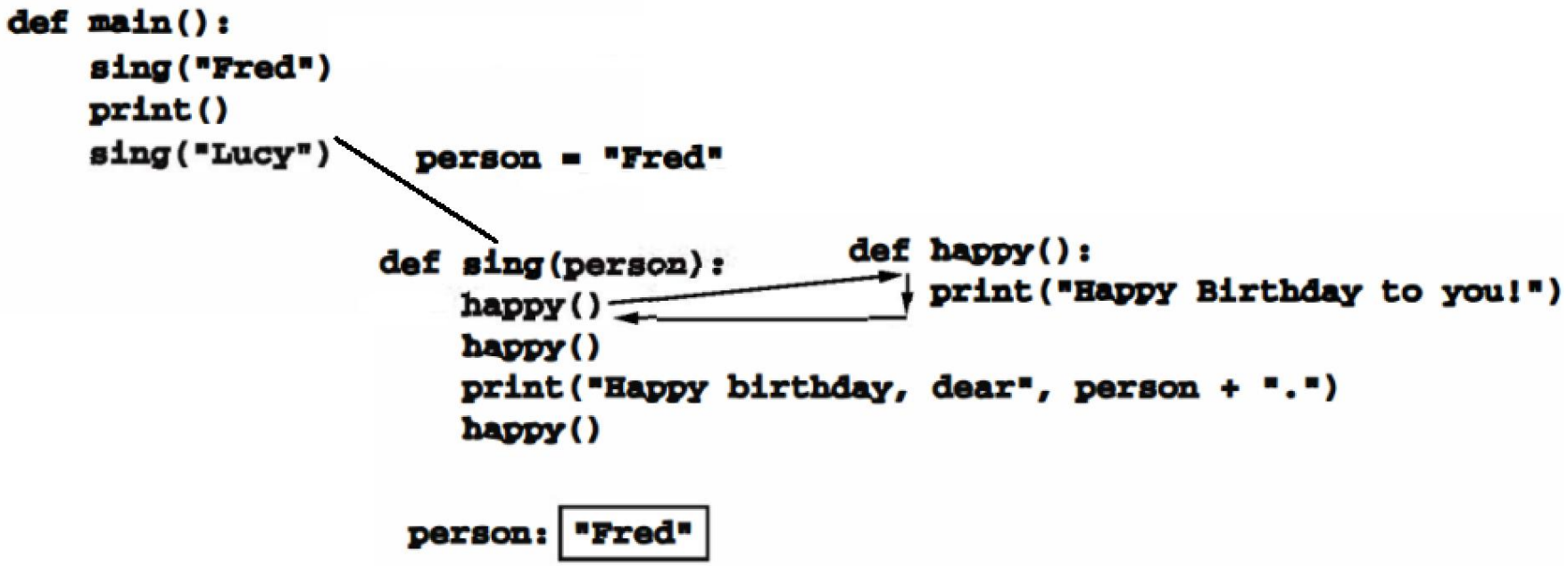
```
sing("Fred")
print()
sing("Lucy")
```



The variable **person** has just been initialized.

Functions and Parameters: The Details

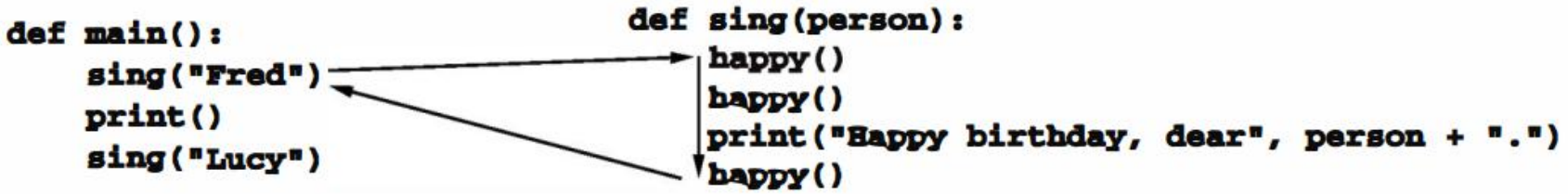
- Python begins executing the body of **`sing`**:



- When Python gets to the **end** of **`sing`**, control returns to **`main`** and continues immediately after the function call.

Functions and Parameters: The Details

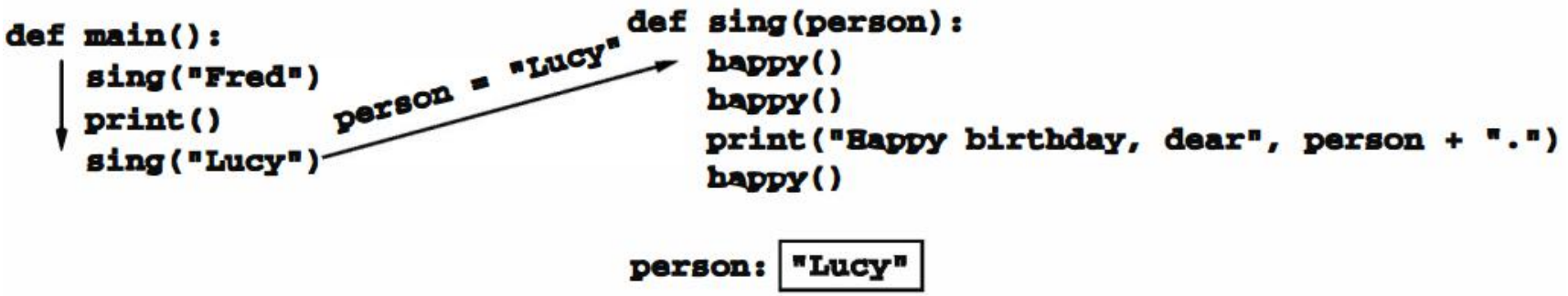
- The completed call to sing

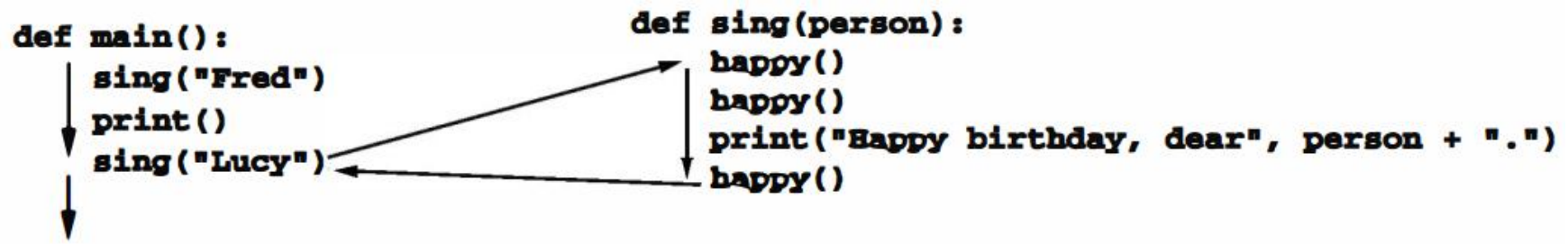


- Notice that the **`person`** variable in **`sing`** has disappeared!
- The memory occupied by local function variables is **reclaimed** when the function exits.
- **Local variables** do **not** retain any values from one function execution to the next.

Functions and Parameters: The Details

- After **`print()`**, python encounters **another call** to **`sing`**, and control transfers to the **`sing`** function.

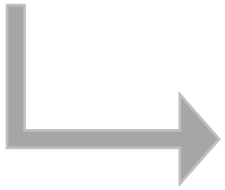




Functions and Paramters: The Details

- As to multiple parameters, the formal and actual parameters are matched up based on ***position.***The first actual parameter is assigned to the first formal parameter, the second is assigned to the second formal parameter, etc.
- For example:

```
drawBar(win, 0, principal)
def drawBar(window, year, height):
```



```
window = win
year = 0
height = principal
```

Functions That Return Values

- We've already seen numerous functions that **return values to the caller**.

```
discRt = math.sqrt(b*b - 4*a*c)
```

- The value **`b*b - 4*a*c`** is the **actual parameter**.
- We say **`sqrt`** *returns* the square root of its argument.

Functions That Return Values

- This function returns the square of a number:

```
def square(x):
    return x ** 2
```

- When Python encounters **`return`**, it exits the function and **returns** control to the point where the function was called.
- The value(s) provided in the **`return`** statement are sent back to the caller as an expression result.

Functions That Return Values

```
>>>
>>> def square(x):
...     return x*x
...
>>> square(3)
9
>>> x=5
>>> y=square(x)
>>> print(y)
25
>>> print (square(x)+square(3))
34
>>>
```

Functions That Return Values

- We can use the square function to **write another function** to calculate the distance between (x_1,y_1) and (x_2,y_2).

```
def distance(p1, p2):
    dist = math.sqrt(square(p2.getX() - p1.getX())
                     + square(p2.getY() - p1.getY())
    return dist
```

- Let's find the example on P189 of the book.

Functions That Return Values

- Let's go back to the Happy Birthday program.

```
# happy2.py

def happy():
    return "Happy Birthday to you!\n"

def verseFor(person):
    lyrics = happy()*2 + "Happy birthday, dear " + person + ".\n" + happy()
    return lyrics

def main():
    for person in ["Fred", "Lucy", "Elmer"]:
        print(verseFor(person))

main()
```

being more elegantand, also more flexible

Functions That Return Values

• We can easily modify the program to **write the results into a file** instead of to the screen.

```
def main():
    outf = open("Happy_Birthday.txt", "w")
    for person in ["Fred", "Lucy", "Elmer"]:
        print(verseFor(person), file=outf)
    outf.close()
```

- **Having functions return values** rather than printing information to the screen gives the caller more choices.

Functions That 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.

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

Functions That Return Values

- When calling this function, use **simultaneous assignment**.
- The values are assigned **based on position**, so *s* gets the first value returned (the sum), and *d* gets the second (the difference).

```
num1, num2 = input("Please enter two numbers (num1, num2) ").split(",")
s, d  = sumDiff(float(num1), float(num2))
print("The sum is", s, "and the difference is", d)
```

Functions That Return Values

- 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`**!

```
# happy2.py
def happy():
    return "Happy Birthday to you!\n"
def verseFor(person):
    lyrics = happy()*2 + "Happy birthday, dear " + person + ".\n" + happy()
    #return lyrics
def main():
    for person in ["Fred", "Lucy", "Elmer"]:
        print(verseFor(person))
main()
```

```
D:\Anaconda3\python.exe C:/Users/J
None
None
None

Process finished with exit code 0
```

Functions that Modify Parameters

- Sometimes, we can **communicate back to the caller** by making changes to the function parameters.
  - Here is a first attempt at such a function:

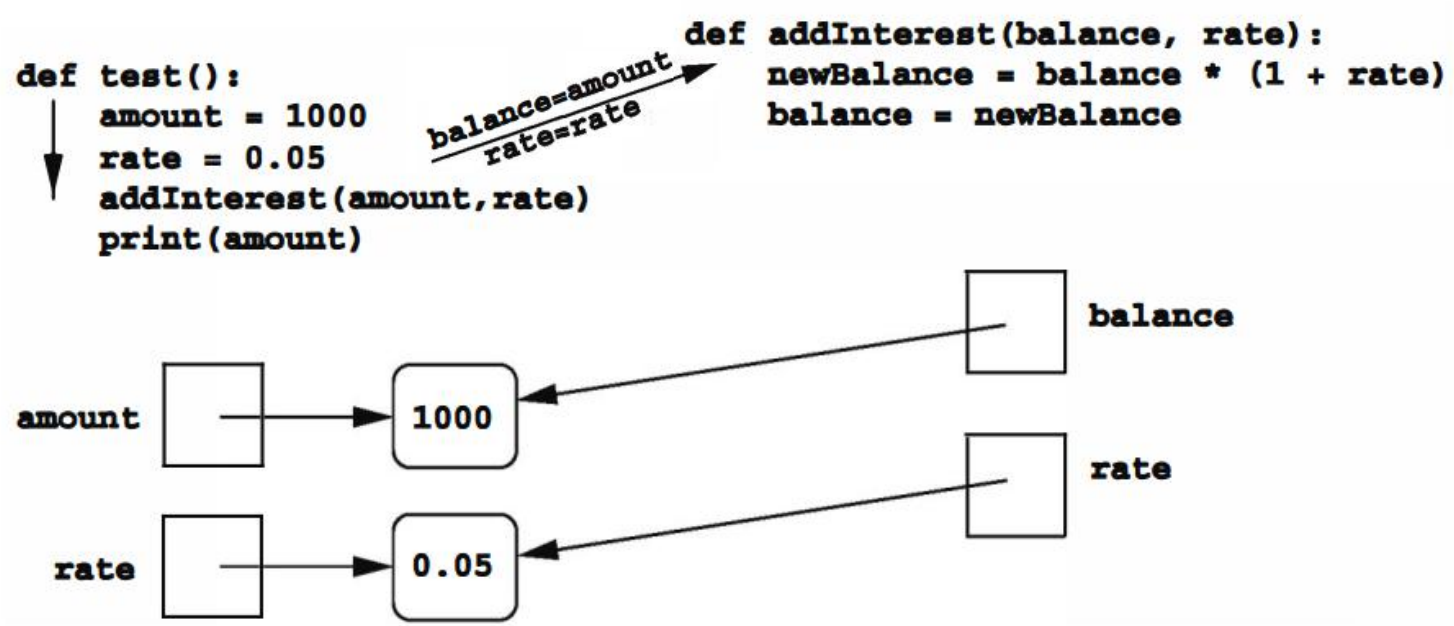
```
# addinterest1.py
def addInterest(balance, rate):
    newBalance = balance * (1+rate)
    balance = newBalance
```

  - Let's try out our function(but, *the output is 1000*):

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

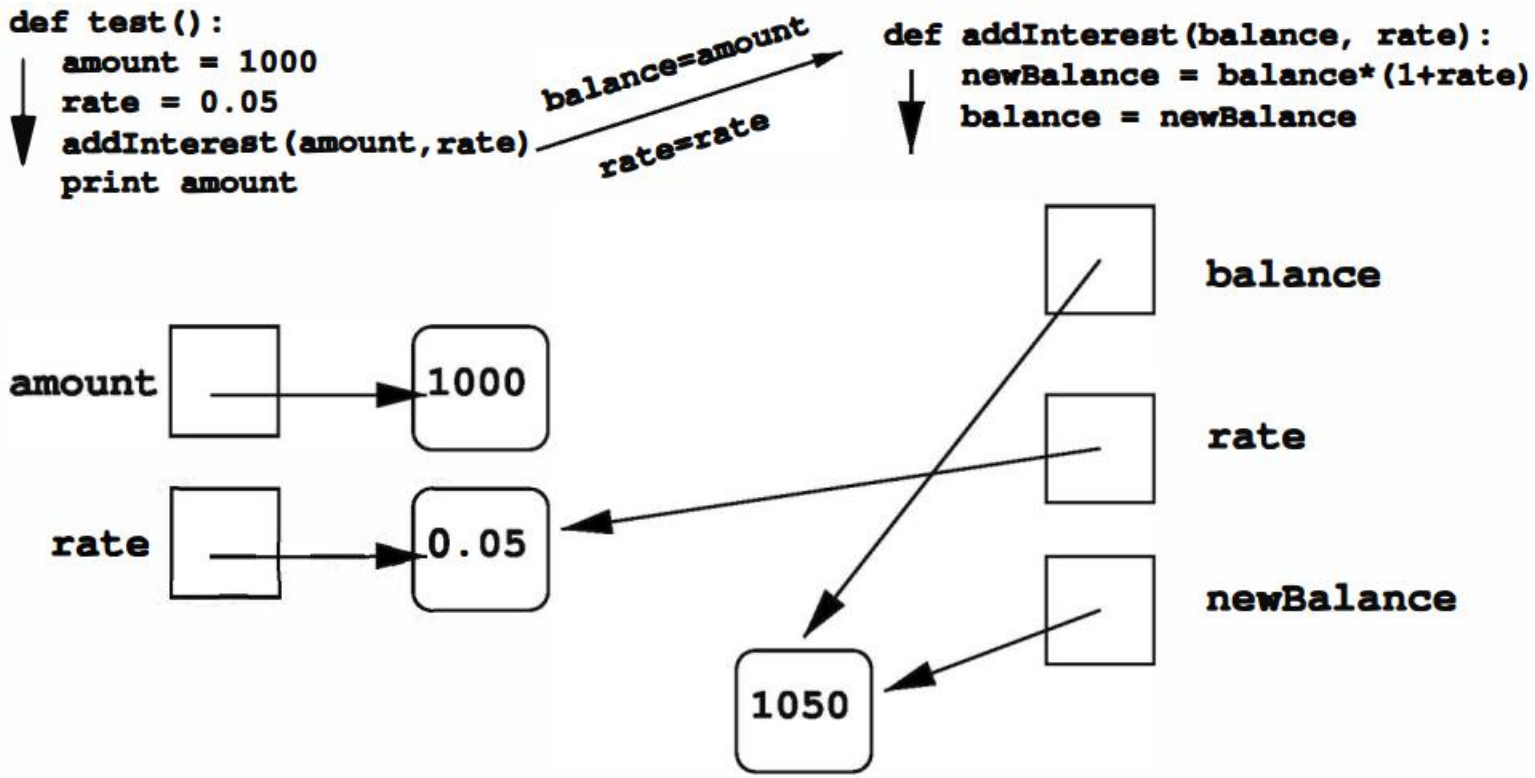
Functions that Modify Parameters

- What went wrong? Nothing!
  - Transfer of control to addinterest



Functions that Modify Parameters

- Assignment of balance



Functions that Modify Parameters

- One alternative would be to change the **`addInterest`** function so that it **returns** the **`newBalance`**.

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

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

Functions that Modify Parameters

- 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

- A more general way to do this would be with **a loop** that goes through positions **0, 1, …, length – 1**.

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

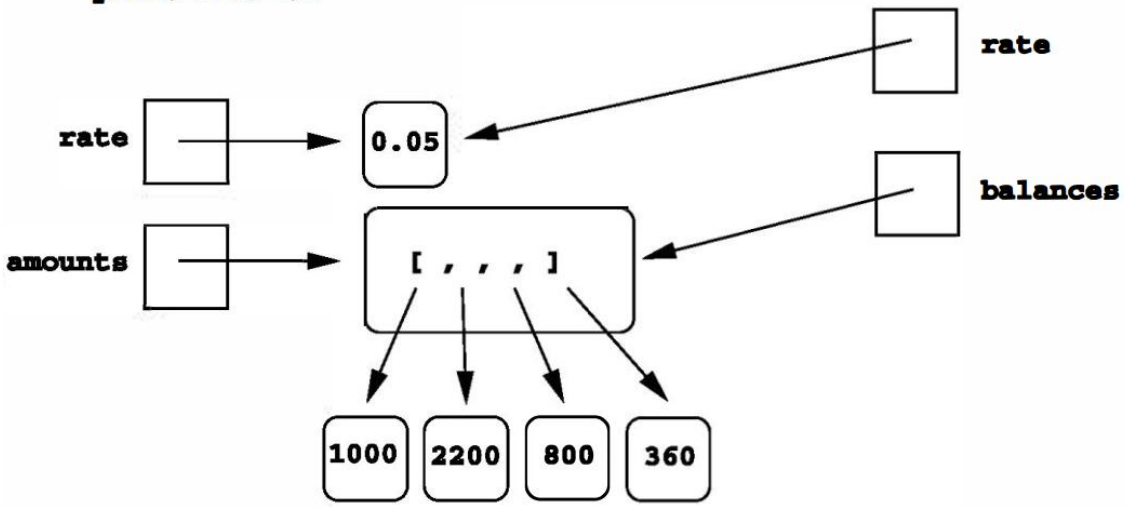
Amounts has been changed!

Functions that Modify Parameters

- Transfer of list parameter to addinterest:

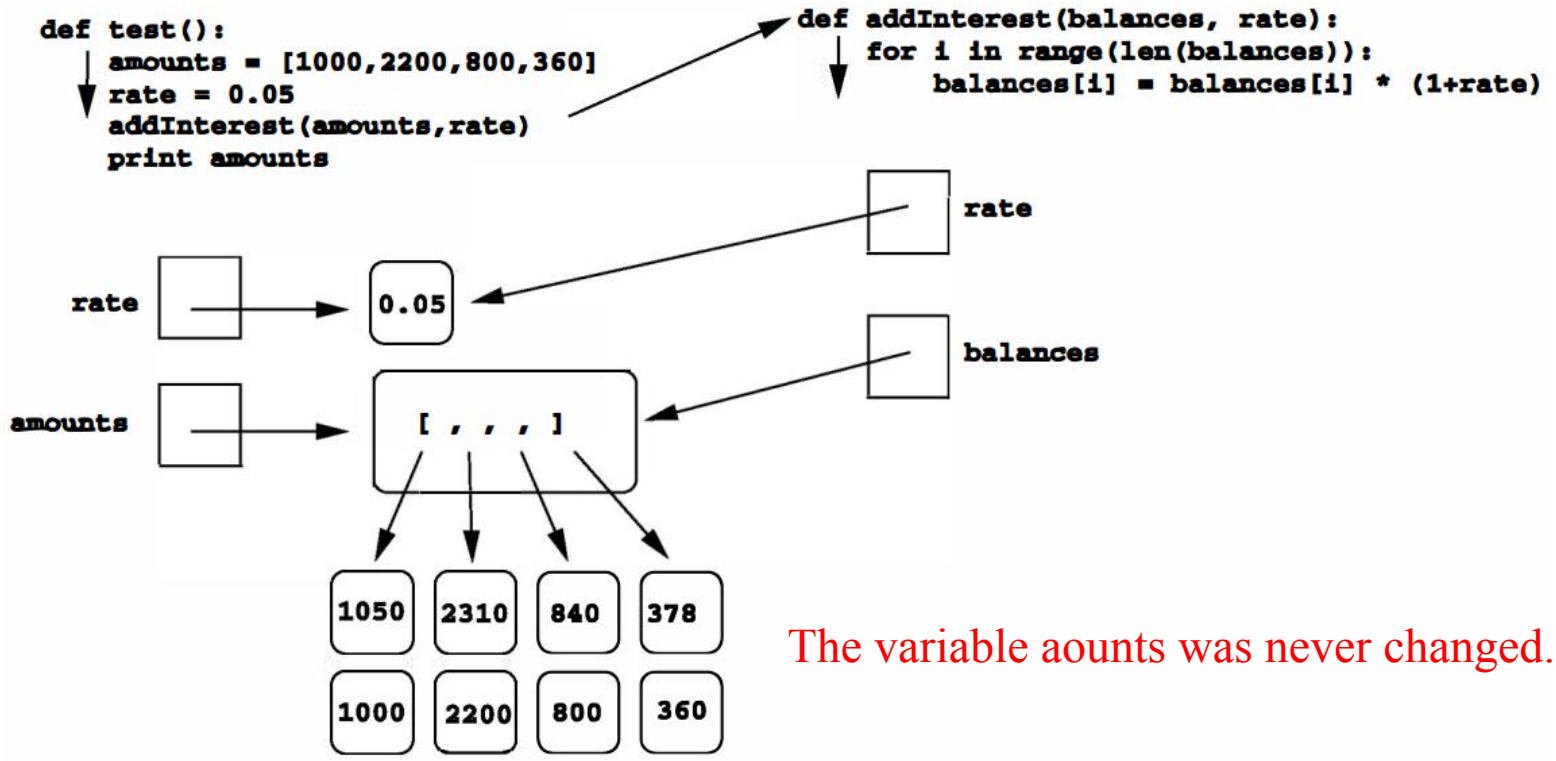
```
def test():
    amounts = [1000,2200,800,360]
    rate = 0.05
    addInterest(amounts,rate)
    print amounts
```

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



Functions that Modify Parameters

- List modifed in addinterest:



The variable aounts was never changed.

Functions that Modify Parameters

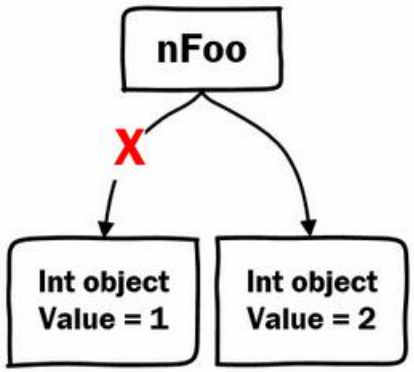
- To summarize: the formal parameters of a function only **receive** the ***values*** **of the actual parameters**. The function **does not have access to** the **variable** that holds the actual parameter.
- Pthon passes all parameters **by value.**But to a **mutable** or **immutable** parameter, it's value will be modified in different ways.

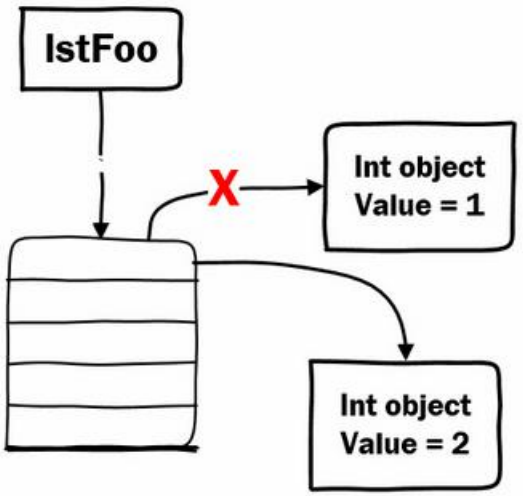
Functions that Modify Parameters

- Mutable & immutable objects:
  - **A mutable object** can be changed after it is created,such as list、dic、set，etc.
  - **An immutable** object can't be changed,for example int、float、str、tuple, etc.

Functions that Modify Parameters

```
>>> nfoo=1
>>> print(id(nfoo))
492400096
>>> nfoo=2
>>> print(id(nfoo))
492400128
>>>
>>> lstFoo=[1]
>>> print(lstFoo)
[1]
>>> print(id(lstFoo))
37089224
>>> lstFoo[0]=2
>>> print(lstFoo)
[2]
>>> print(lstFoo)
[2]
>>> print(id(lstFoo))
37089224
>>>
```





Functions that Modify Parameters

- Somthing important!
  - Example 1:

```
1  #处理多个银行账户的余额信息
2  def addInterest(balances, rate):
3      for i in range(len(balances)):
4          balances[i] = balances[i] * (1+rate)
5  def test():
6      amounts = [1000, 105, 3500, 739]
7      rate = 0.05
8      addInterest(amounts, rate)
9      print(amounts)
10
11 test()
```

To **mutable** parameters, the value will be **changed directly**.

  - Outputs:

```
1  [1050.0, 110.25, 3675.0, 775.95]
```

Functions that Modify Parameters

- Analysis this:

```
# 处理多个银行账户的余额信息
def addInterest(balances, rates):
print()
print("第二处", id(balances))
    for i in range(len(balances)):
        balances[i]= balances[i]*(1+rates)
        print()
        print("第三处",id(balances))
def test():
    amounts = [1000,105,3500,739]
    print()
    print("第一处",id(amounts))
    rate = 0.05
    addInterest(amounts, rate)
    print()
    print(amounts)
    print()
    print("第四处",id(amounts))
test()
```

```
第一处 41203656

第二处 41203656

第三处 41203656

第三处 41203656

第三处 41203656

第三处 41203656

[1050.0, 110.25, 3675.0, 775.95]

第四处 41203656
```

Functions that Modify Parameters

- Example 2:

```
 # 计算单个银行账户余额
def addinterest(balance, rate):
    print("第二处", id(balance))
    newBalance = balance * (1 + rate)
    print()
    print("第三处", id(balance))
    print()
    print("第四处", id(newBalance))
    return newBalance


def main():
    amount = 1000
    print("第一处", id(amount))
    print()
    rate = 0.05
    amount = addinterest(amount, rate)
    print()
    print("第五处", id(amount))
    print()
    print(amount)
    print("第六处", id(amount))
```

Functions that Modify Parameters

- Ouputs:

```
1  第一处 33533648
2
3  第二处 33533648
4
5  第三处 33533648
6
7  第四处 33563344
8
9  第五处 33563344
10
11 1050.0
12 第六处 33563344
```

To **immutable** parameters, a new object will be build and returned.

Functions that Modify Parameters

- Example 3:

```
1 def change(val):
2     newval = [10]
3     val= val + newval # val=val+[10] is not serious
4
5 nums = [0, 1]
6 change(nums)
7 print(nums)
```

- The ouput:

```
1 [0, 1]
```

With "=" ,the **mutable** parameter "val" , point to a new memory address.

Functions that Modify Parameters

- Change the program like this, then it will works!

```
1 def change(val):
2     val.append(10)
3 nums = [0, 1]
4 change(nums)
5 print(nums)
```

Functions and Program Structure

- As the **algorithms** you design get **increasingly complex**, it gets more and more difficult to make sense out of the programs.
- One way is to break an algorithm down into **smaller subprograms**, each of which makes sense on its own.
- Let's go to the **futval.py** again.

Functions and Program Structure

```
def main():
    # Introduction
    print("This program plots the growth of a 10-year investment.")

    # Get principal and interest rate
    principal = float(input("Enter the initial principal: "))
    apr = float(input("Enter the annualized interest rate: "))

    # Create a graphics window with labels on left edge
    win = GraphWin("Investment Growth Chart", 320, 240)
    win.setBackground("white")
    win.setCoords(-1.75,-200, 11.5, 10400)
    Text(Point(-1, 0), ' 0.0K').draw(win)
    Text(Point(-1, 2500), ' 2.5K').draw(win)
    Text(Point(-1, 5000), ' 5.0K').draw(win)
    Text(Point(-1, 7500), ' 7.5k').draw(win)
```

Functions and Program Structure

```
    Text(Point(-1, 10000), '10.0K').draw(win)

    # Draw bar for initial principal
    drawBar(win, 0, principal)

    # Draw a bar for each subsequent year
    for year in range(1, 11):
        principal = principal * (1 + apr)
        drawBar(win, year, principal)

    input("Press <Enter> to quit.")
    win.close()

main()
```

Functions and Program Structure

- We can make this program **more readable** by using a **value returning function**.

```
def createLabeledWindow():
    # Returns a GraphWin with title and labels drawn
    window = GraphWin("Investment Growth Chart", 320, 240)
    window.setBackground("white")
    window.setCoords(-1.75,-200, 11.5, 10400)
    Text(Point(-1, 0), ' 0.0K').draw(window)
    Text(Point(-1, 2500), ' 2.5K').draw(window)
    Text(Point(-1, 5000), ' 5.0K').draw(window)
    Text(Point(-1, 7500), ' 7.5k').draw(window)
    Text(Point(-1, 10000), '10.0K').draw(window)
    return window
```

Functions and Program Structure

```
def main():
    print("This program plots the growth of a 10-year investment.")

    principal = input("Enter the initial principal: ")
    apr = input("Enter the annualized interest rate: ")

    win = createLabeledWindow()
    drawBar(win, 0, principal)
    for year in range(1, 11):
        principal = principal * (1 + apr)
        drawBar(win, year, principal)

    input("Press <Enter> to quit.")
    win.close()
```

It becomes nearly self-documenting