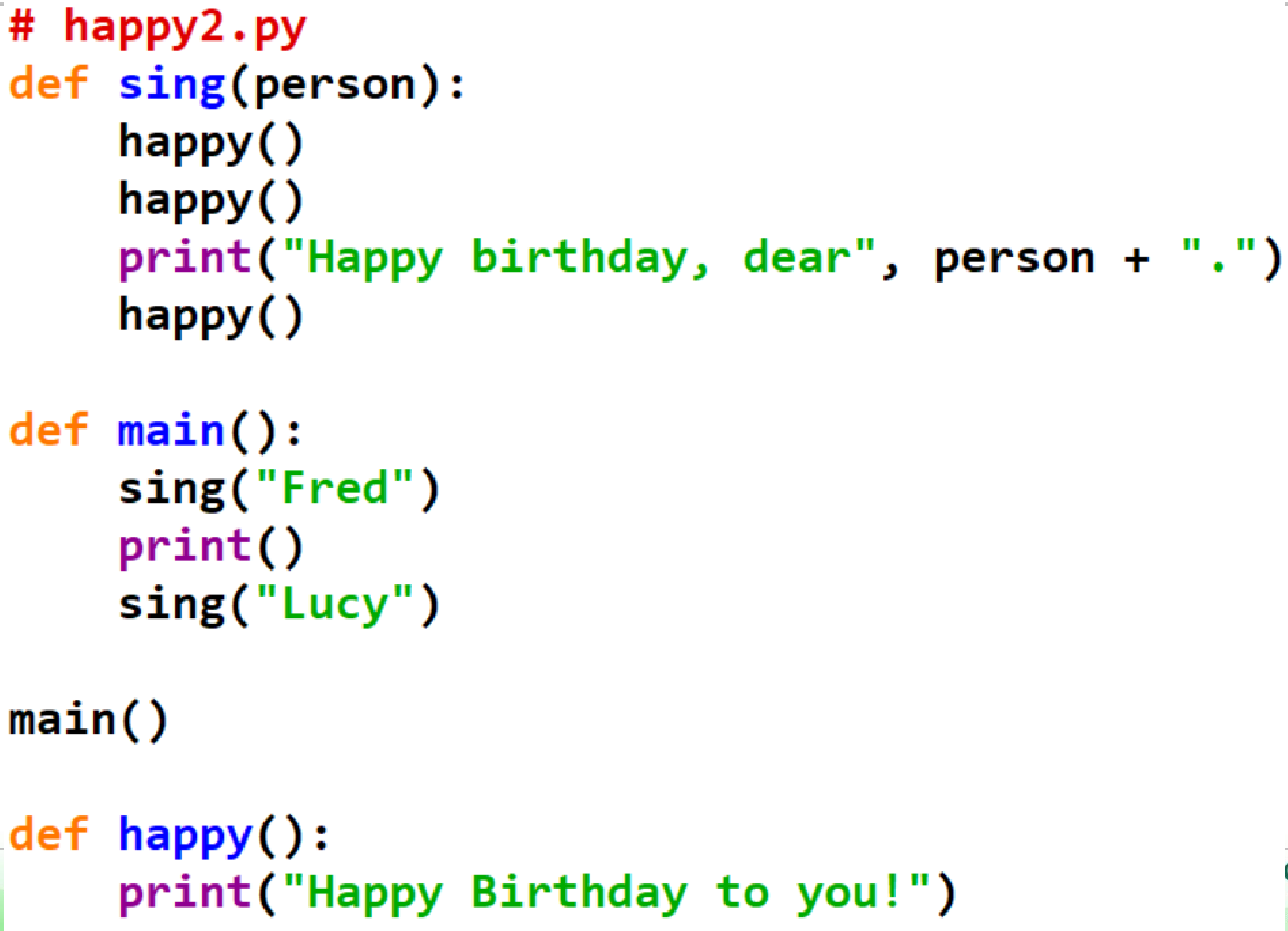
Debugging Exercise



```python
# happy2.py
def sing(person):
    happy()
    happy()
    print("Happy birthday, dear", person + ".")
    happy()

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

main()

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

Functions and Variable Scope

- Global variables are those defined outside of any function definitions

```
>>> H = 24  # global constant
>>> def test():
...     print('There are', H, 'hours in a day.')
>>> test()
There are 24 hours in a day.
```

Functions and Variable Scope

- The variables used inside of a function are *local* to that function, even if they have the same name as variables outside of the function

```
>>> i = 1
>>> def test():
...     i = 5
...     print(i, 'in test()')
>>> test()
5 in test()
>>> print(i, 'global')
1 global
```

Functions and Parameters

- Function definition syntax

```
def <name>(<formal_parameters>):
    <body>
```

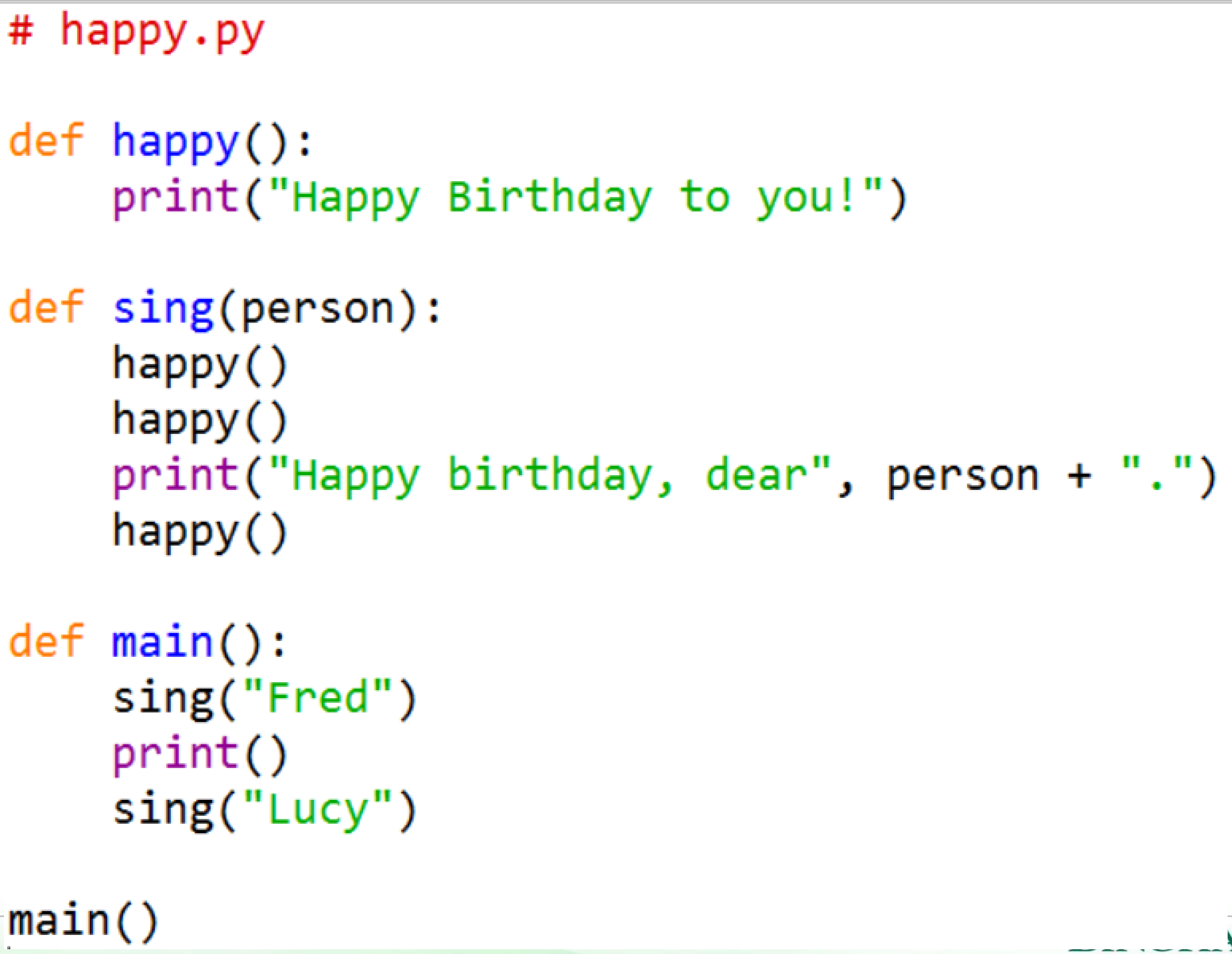
- A function call

```
<name>(<actual_parameters>)
```

Functions and Parameters

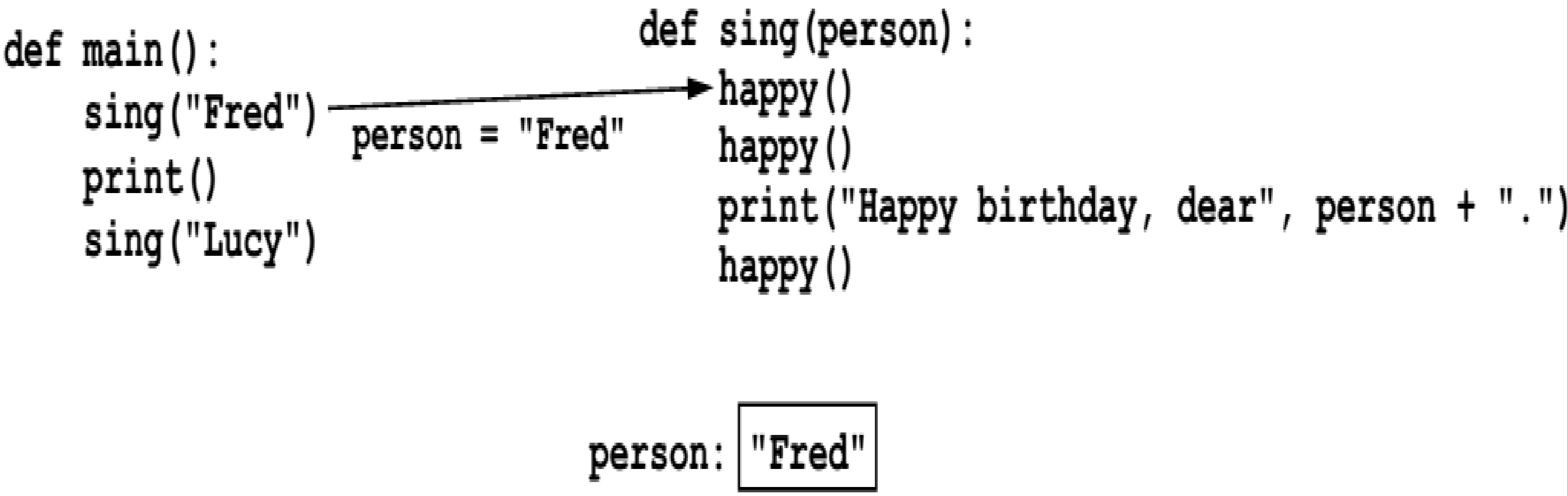
- A function call involves four steps:
  - The calling program suspends execution at the point of the call
  - The values of the actual parameters will be assigned to the formal parameters
  - The body of the function is executed
  - Returns to the point just after where the function was called

Functions and Parameters



```
# happy.py

def happy():
    print("Happy Birthday to you!")

def sing(person):
    happy()
    happy()
    print("Happy birthday, dear", person + ".")
    happy()

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

main()
```

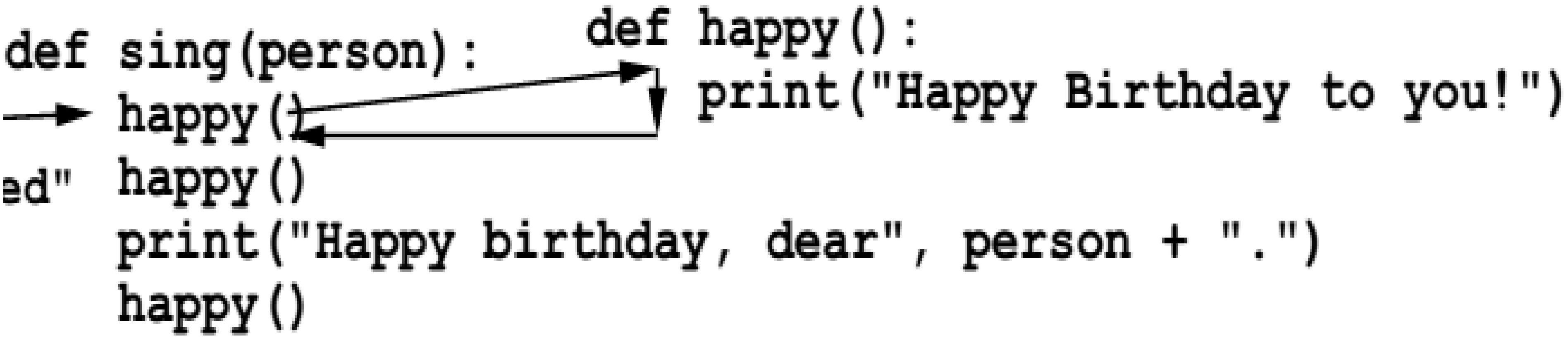
Functions and Parameters



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

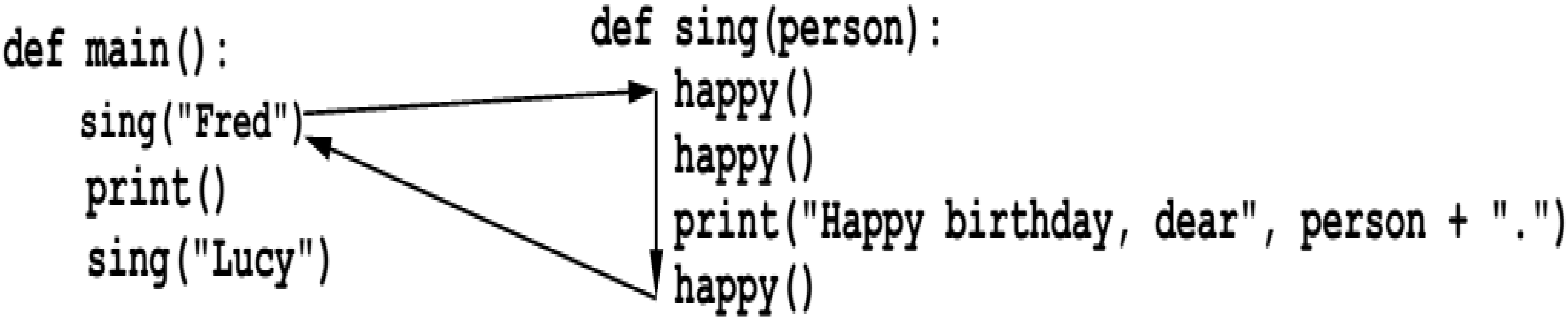
```
def sing(person):
    happy()
    happy()
    print("Happy birthday, dear", person + ".")
    happy()
```

Functions and Parameters



```
def sing(person):          def happy():
    happy()                    print("Happy Birthday to you!")
ed" happy()
    print("Happy birthday, dear", person + ".")
    happy()
```

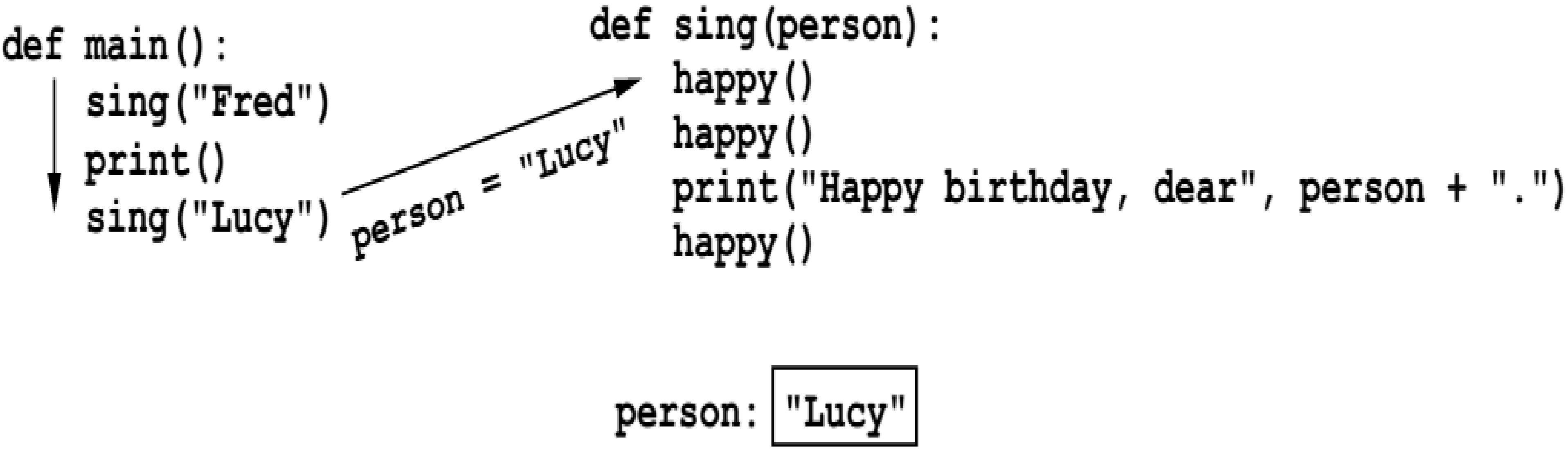
Functions and Parameters



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

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

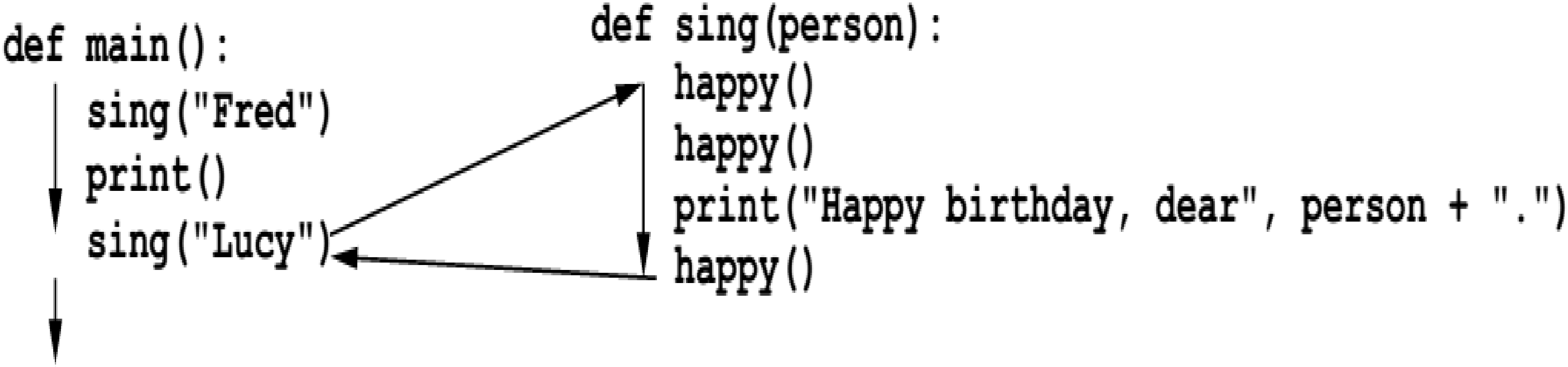
Functions and Parameters



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

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

Functions and Parameters



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

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

Functions and Parameters

- If there are multiple parameters, the formal and actual parameters will match based on position:

```
>>> def calsum(a, b, c):
...     print(a + b + c)
>>> calsum(1, 2, 3)
6
```

Functions and Parameters

- A *default parameter* is a parameter that assumes a default value, if a value is not provided in the function call for that parameter

```
>>> def calsum(a, b, c=3):
...     print(a + b + c)

>>> calsum(1, 2)
6
>>> calsum(1, 2, 4)
7
```

Functions and Parameters

- The default parameter must be put at the end

```
>>> def calsum(a, b=2, c):
...     print(a + b + c)
...
File "<stdin>", line 1
SyntaxError: non-default argument follows default
argument
```

Functions and Parameters

- A *variable-length parameter* is used to pass a variable number of parameters to a function

```
>>> def calsum(a, b, *var):
...     sum = a + b
...     for v in var:
...         sum = sum + v
...     print(sum)

>>> calsum(1, 2)
3
>>> calsum(1, 2, 3, 4)
10
```

Functions Docstrings

- A docstring is used, like a comment, to document a specific segment of code
- It immediately follows the function definition

Functions Docstrings

```
>>> def area(base, height):
...     """Calculate the area of a triangle."""
...     print(base * height / 2)

>>> help(area)
area(base, height)
    Calculate the area of a triangle.
```

Functions That Return Values

- Some functions return values to the caller

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

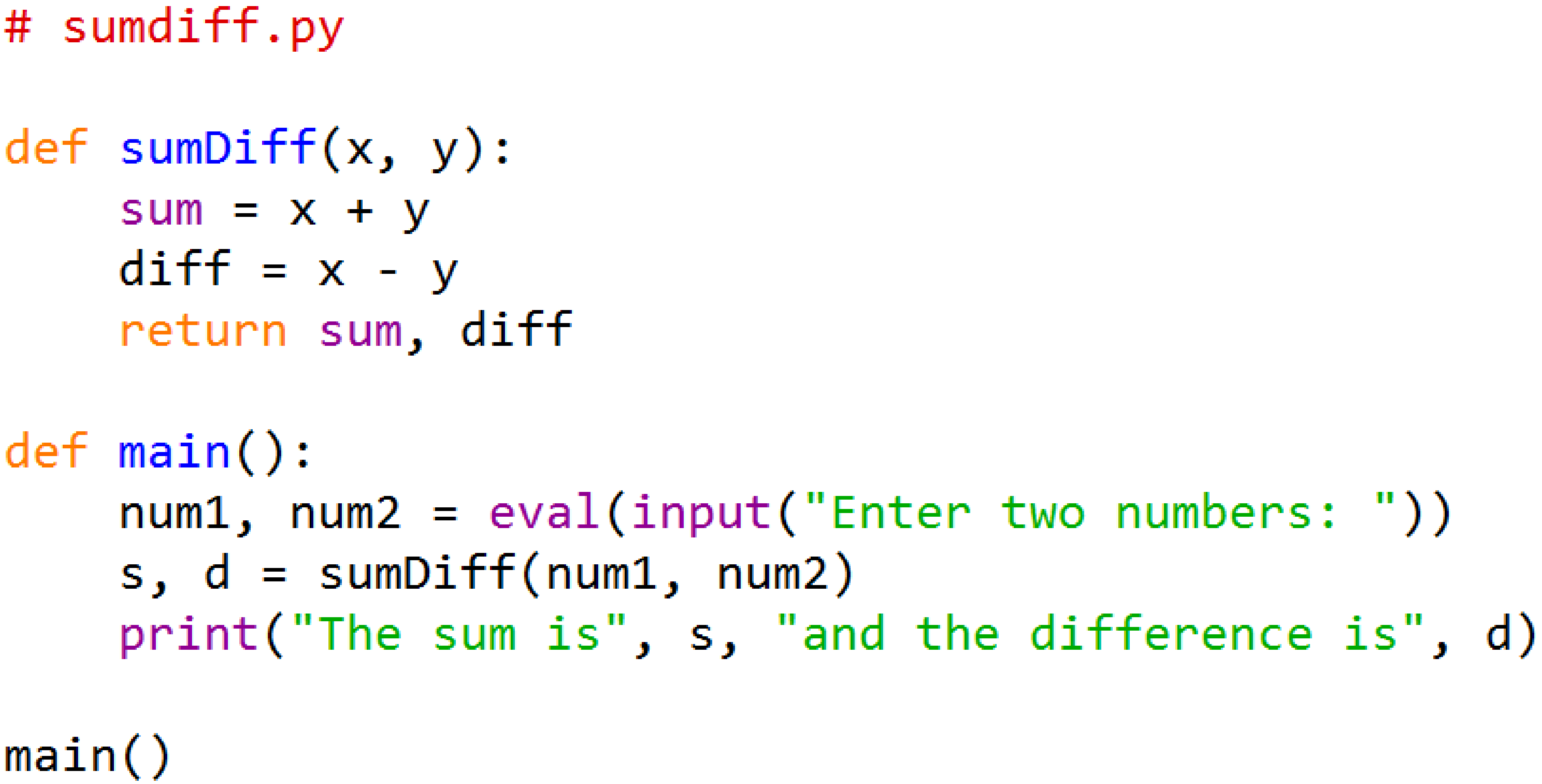
Functions That Return Values

```
>>> def square(x):
...     return x*x

>>> x = 5
>>> y = square(x)
>>> y
25
```

Functions That Return Values

- Use simultaneous assignment to return more than one value

Functions That Return Values



```python
# sumdiff.py

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

def main():
    num1, num2 = eval(input("Enter two numbers: "))
    s, d = sumDiff(num1, num2)
    print("The sum is", s, "and the difference is", d)

main()
```

Functions That Return Values

- Run the program:

```
Enter two numbers: 3, 4
The sum is 7 and the difference is -1
```

Functions That Return Values

- All Python functions return a value, whether they contain a return statement or not
- Functions without a return statement will send back a special object, denoted by None

```
>>> def test():
...     print("No return statement")
>>> a = test()
No return statement
>>> print(a)
None
>>> type(a)
<class 'NoneType'>
```

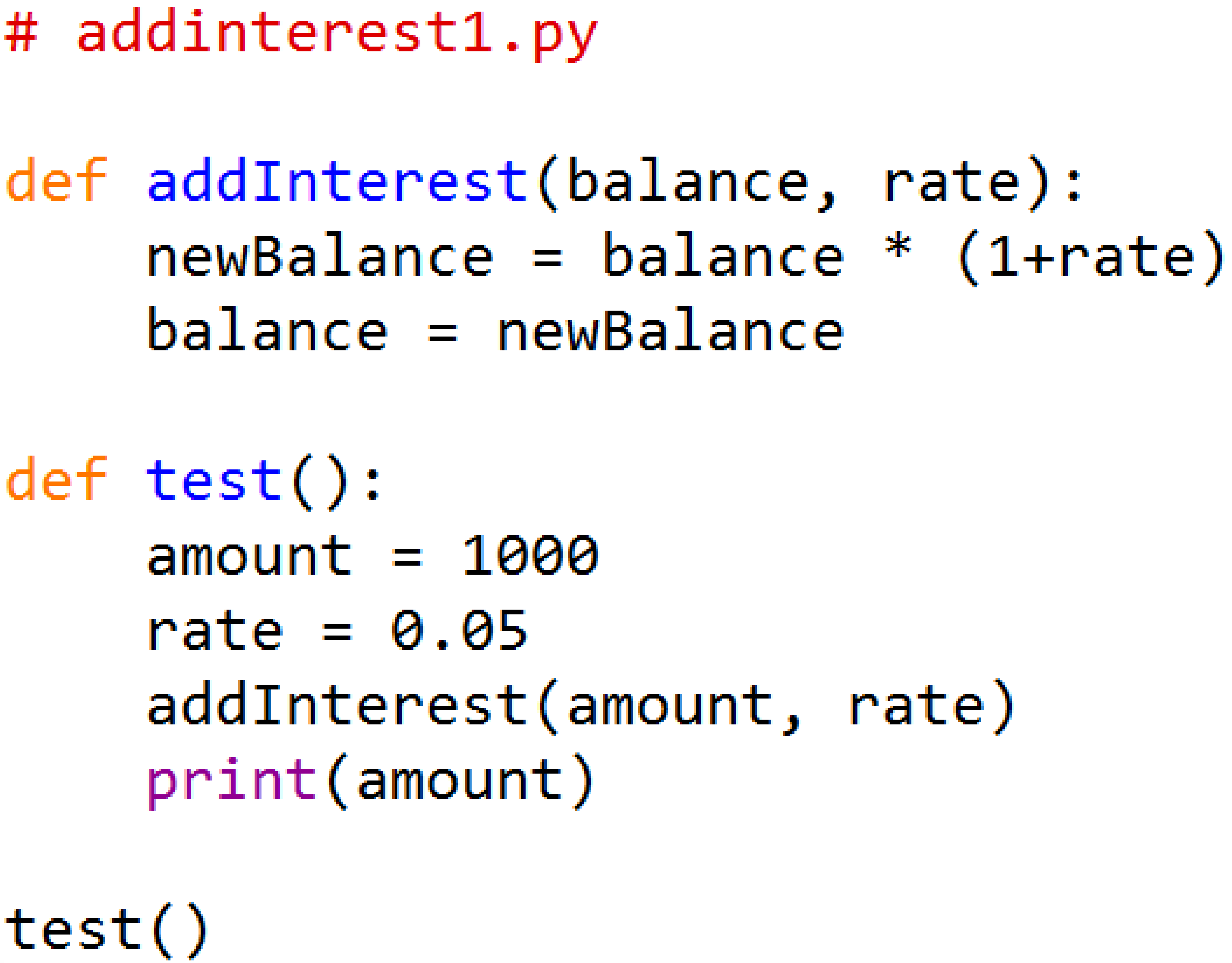
Functions that Modify Parameters

- Another way of communicating back to the caller is by making changes to the function parameters themselves

Functions that Modify Parameters

- The program `addinterest1.py` tries to accumulate interest on a bank account
- For example, a 5% interest added to the principal 1000 returns 1050

Functions that Modify Parameters

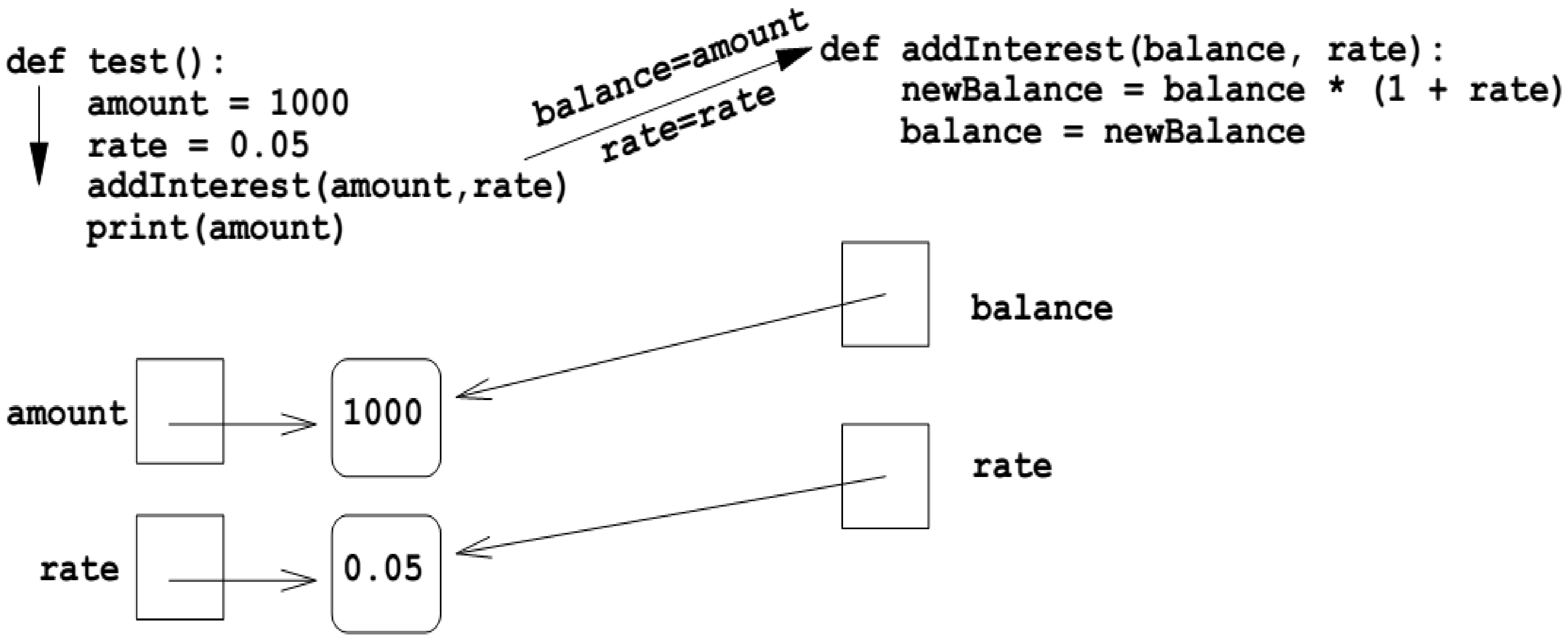


```
# addinterest1.py

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

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

test()
```

Functions that Modify Parameters

- Run the program:
  <u>???</u>

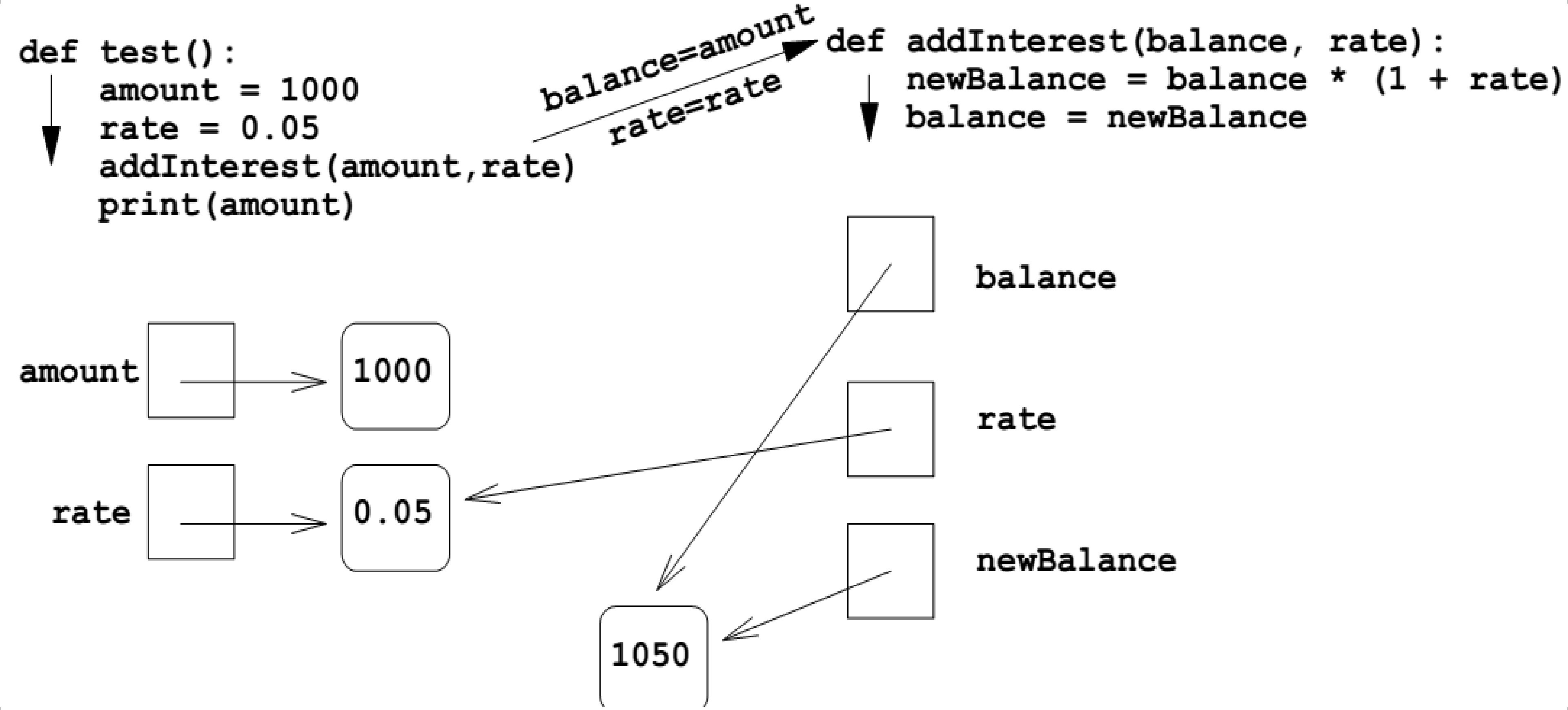
Functions that Modify Parameters



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

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

Functions that Modify Parameters



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

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

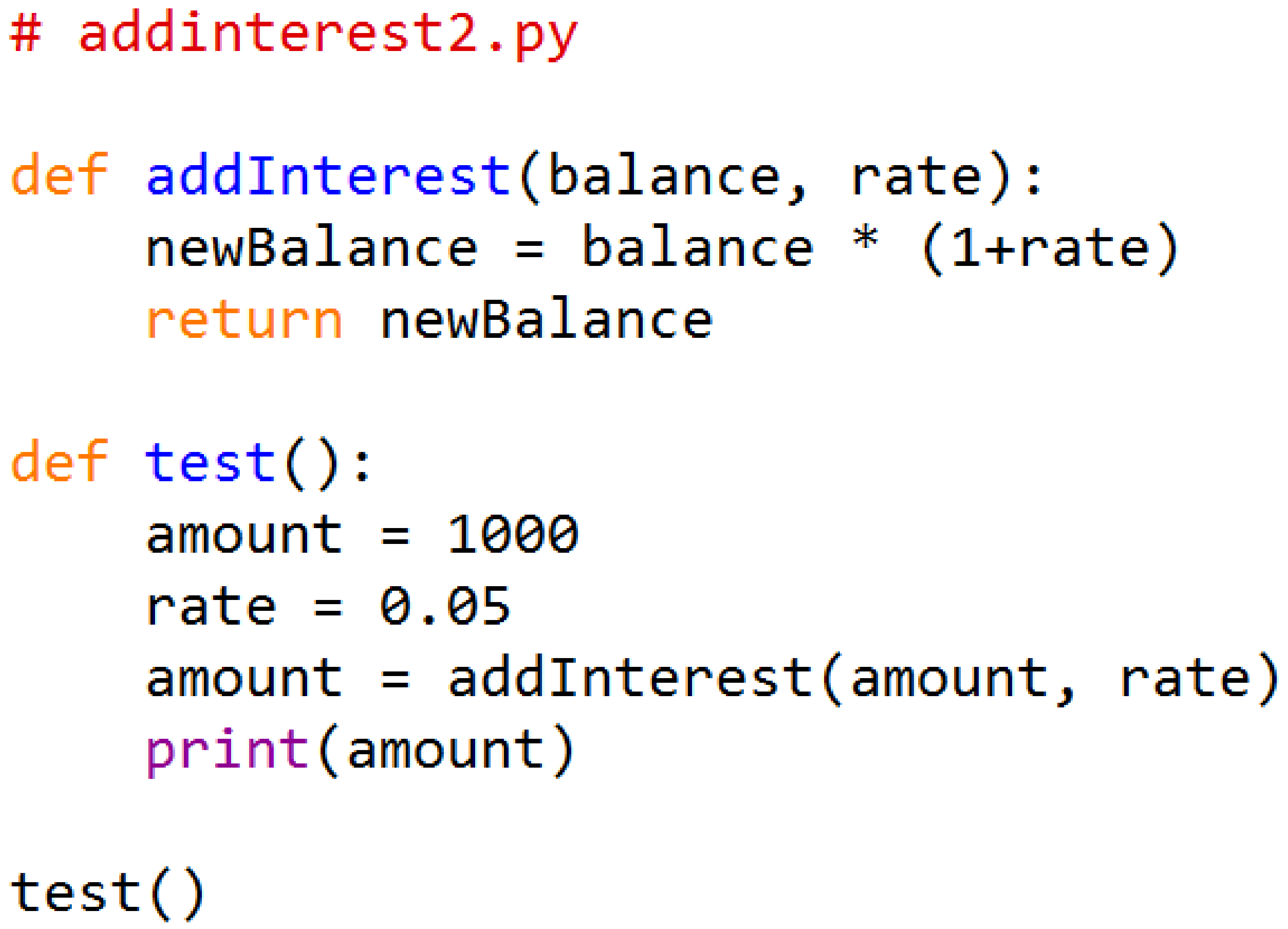
Functions that Modify Parameters

- The formal parameters are *assigned the values* of the actual parameters
- Python passes parameters *by assignment*

Functions that Modify Parameters

- We need to change the `addInterest` function so that it returns the `newBalance`

Functions that Modify Parameters



```
# addinterest2.py

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

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

test()
```

Functions that Modify Parameters

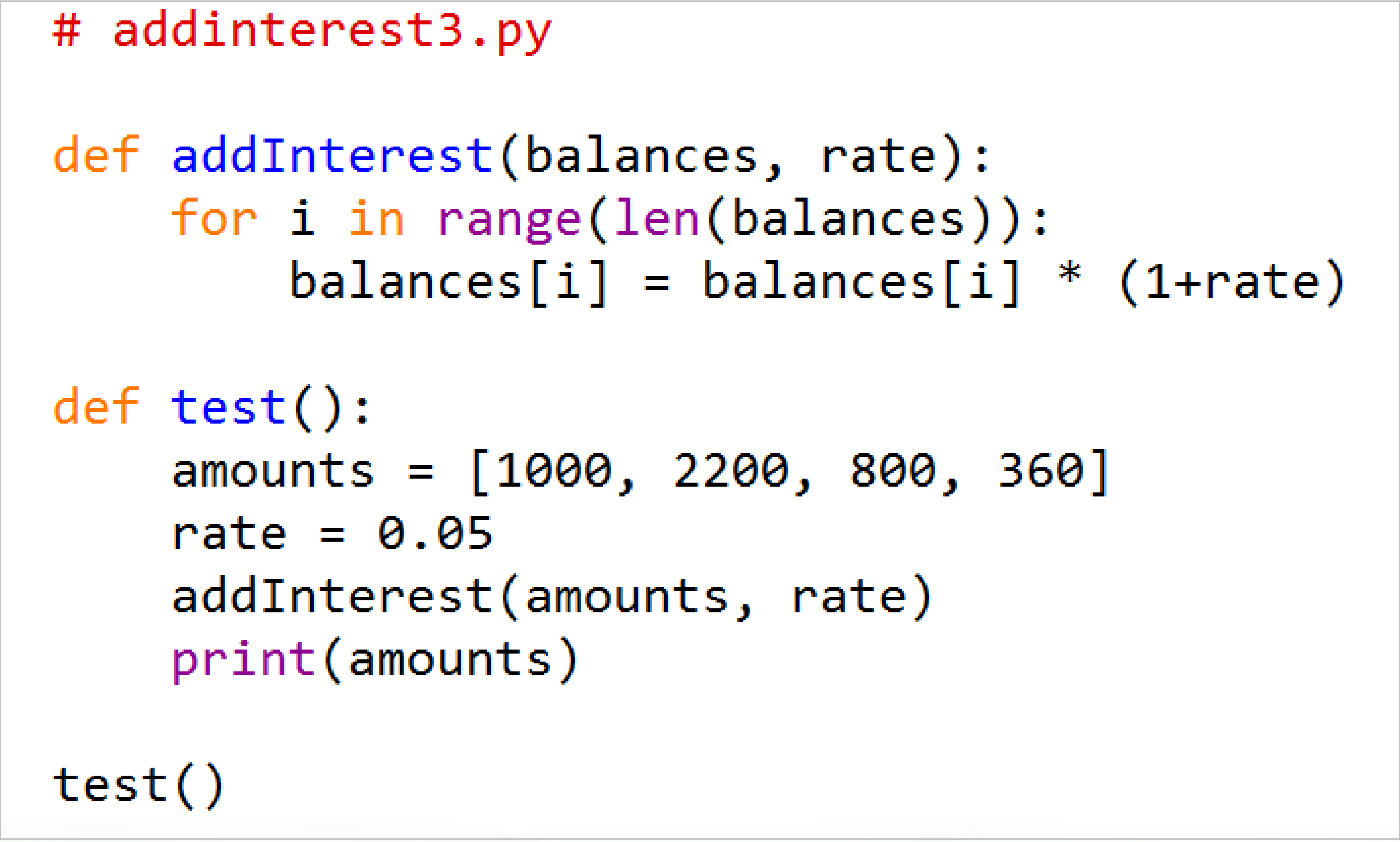
- Run the program:

```
1050.0
```

Functions that Modify Parameters

- Write a program that deals with many accounts
- Store the account balances in a list
- Add the accrued interest to each of the balances in the list

Functions that Modify Parameters



```python
# addinterest3.py

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

- Our original code had these values:

```
[1000, 2200, 800, 360]
```

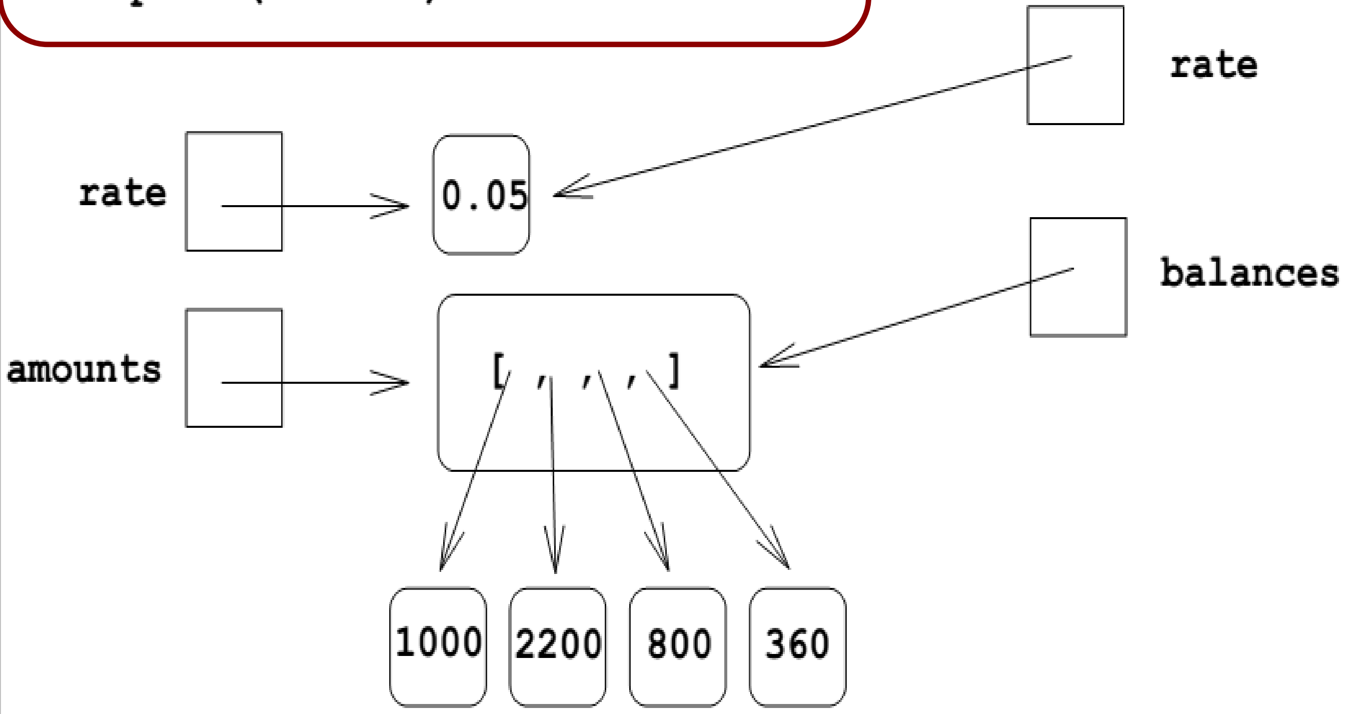
- After we run the program, it returns:

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

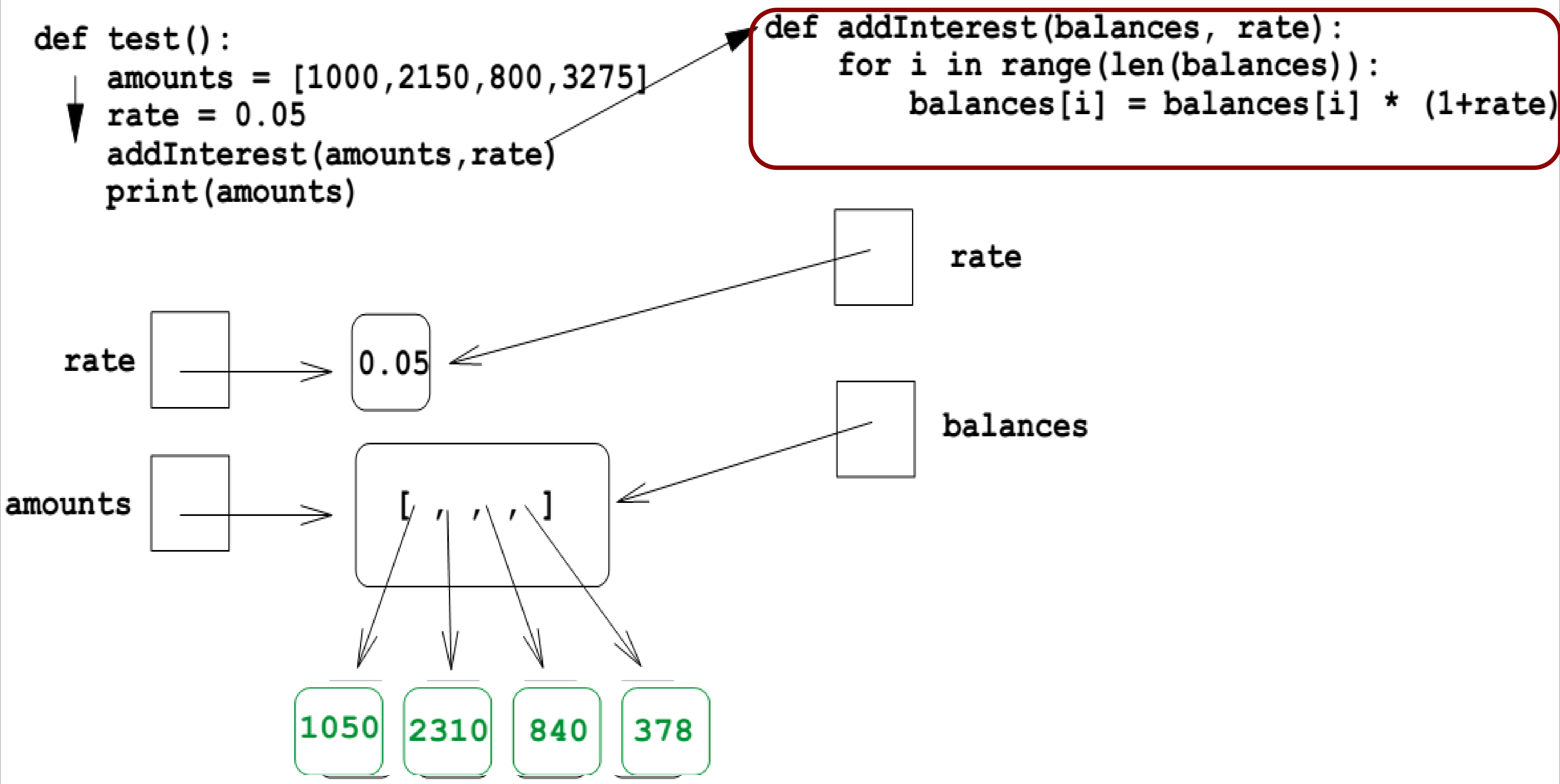
Functions that Modify Parameters

```
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



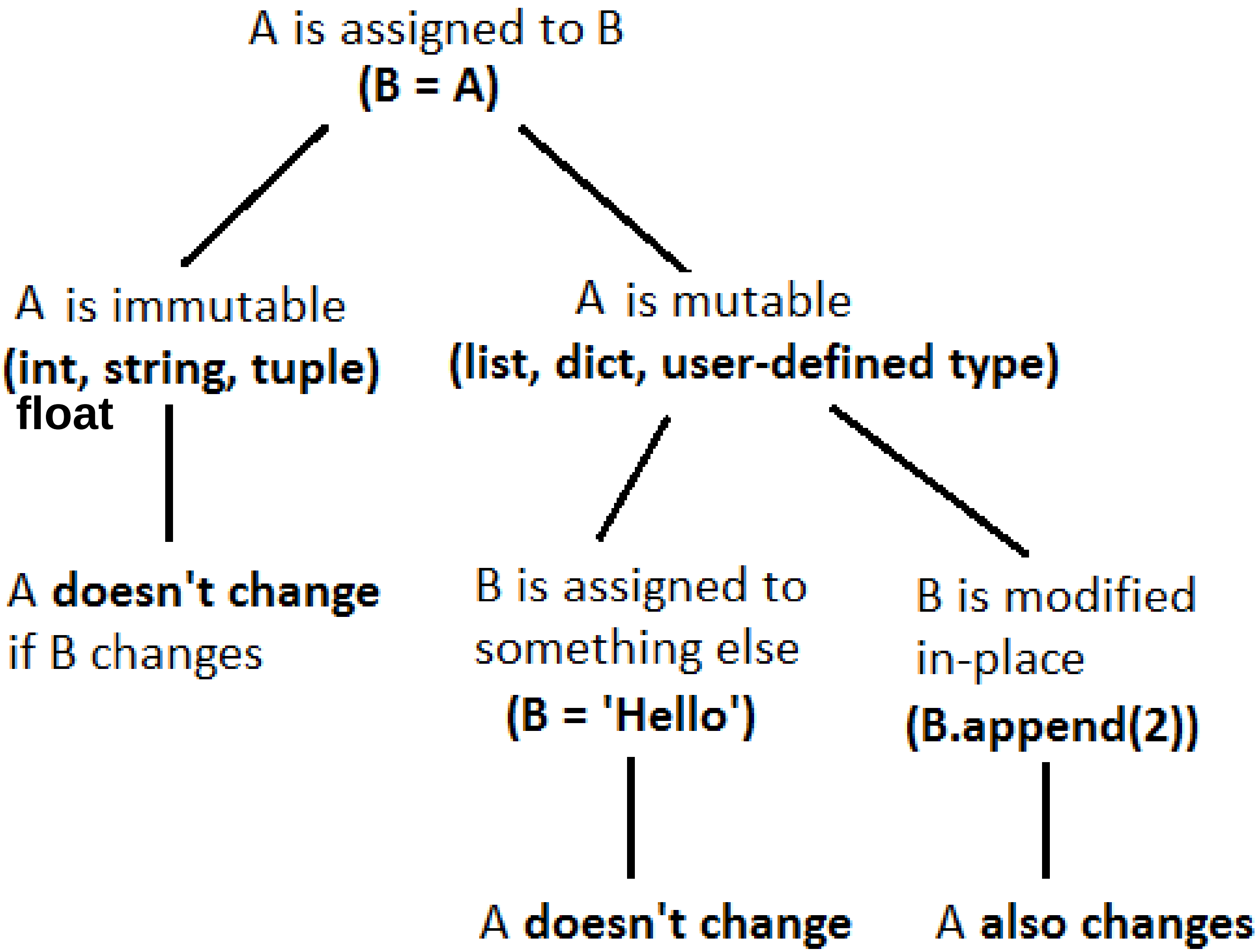
```
def test():
    amounts = [1000,2150,800,3275]
    rate = 0.05
    addInterest(amounts,rate)
    print(amounts)
```

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

Functions that Modify Parameters

- If the value of the variable is a mutable object (like a list), then in-place changes to the object *will* be visible to the calling program

Functions that Modify Parameters



A is assigned to B

(B = A)

A is immutable

(int, string, tuple)

float

A is mutable

(list, dict, user-defined type)

A **doesn't change**

if B changes

B is assigned to

something else

(B = 'Hello')

B is modified

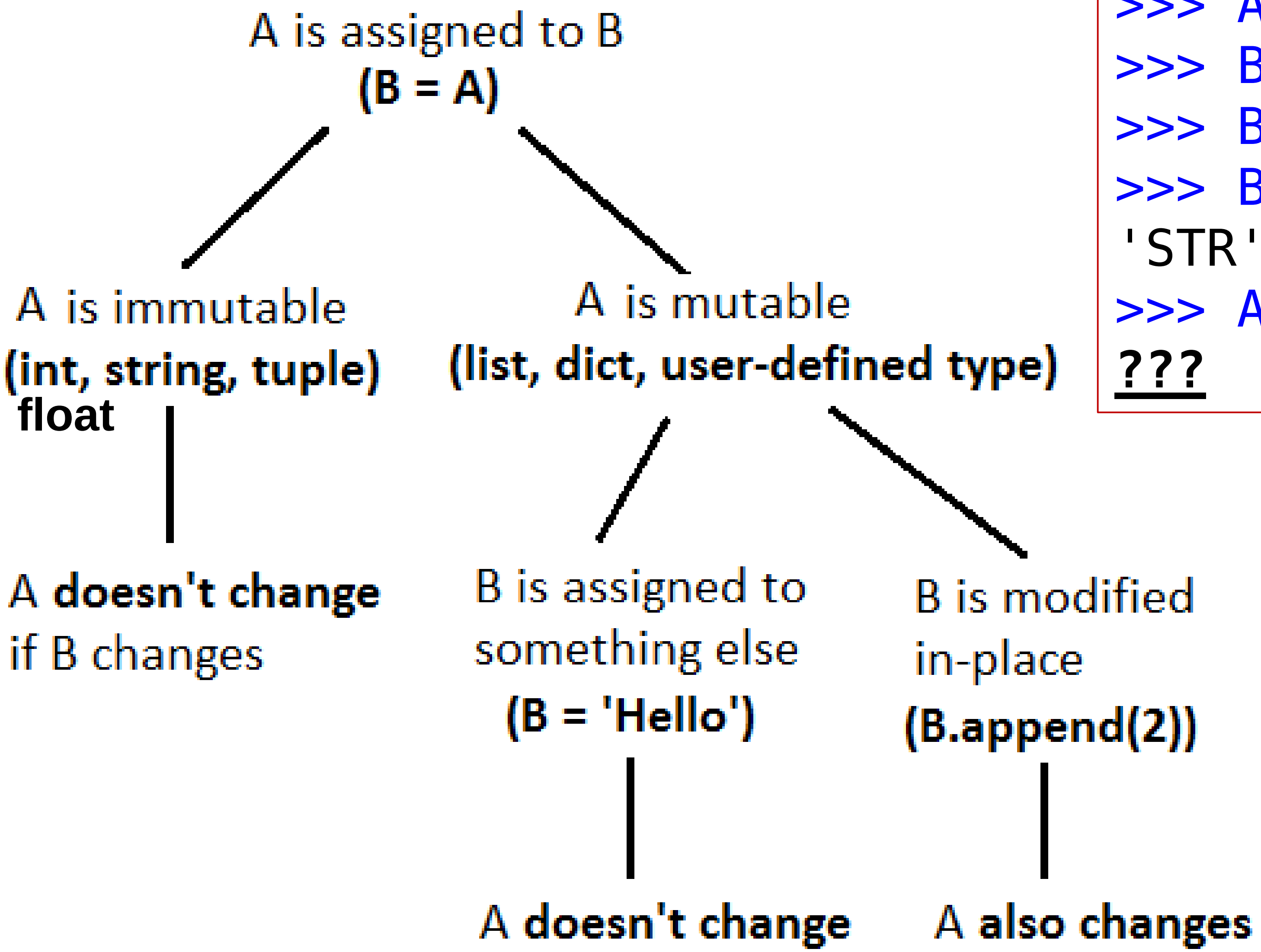
in-place

(B.append(2))

A **doesn't change**

A **also changes**

Functions that Modify Parameters



A is assigned to B
(B = A)

A is immutable
(int, string, tuple)
float

A is mutable
(list, dict, user-defined type)

A doesn't change
if B changes

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

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

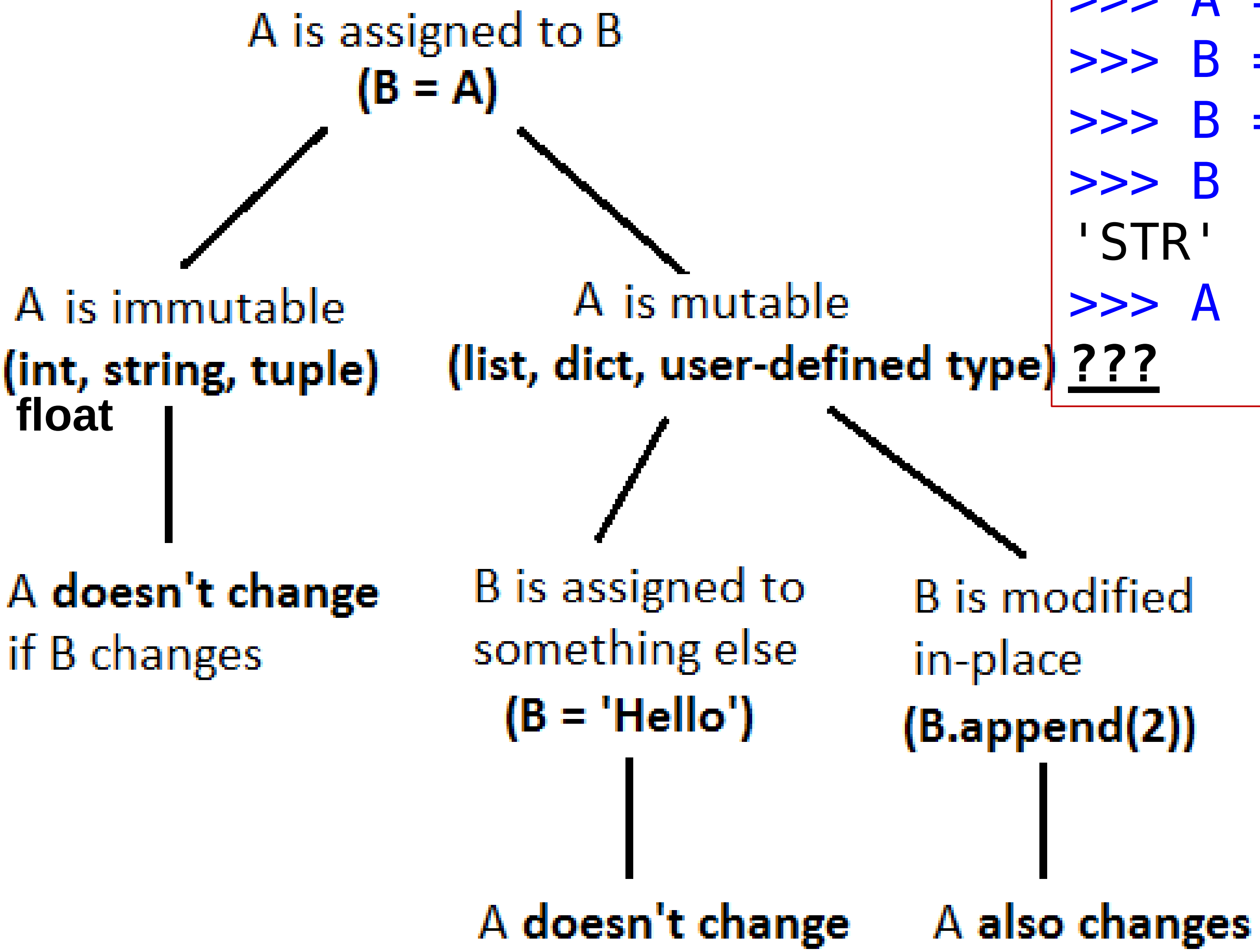
A doesn't change

A also changes

```
>>> A = 1
>>> B = A
>>> B = 'STR'
>>> B
'STR'
>>> A
???
```

Functions that Modify Parameters

```
>>> A = [1, 2]
>>> B = A
>>> B = 'STR'
>>> B
'STR'
>>> A
???
```



A is assigned to B

(B = A)

A is immutable

(int, string, tuple)

float

A is mutable

(list, dict, user-defined type)

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

B is assigned to something else

(B = 'Hello')

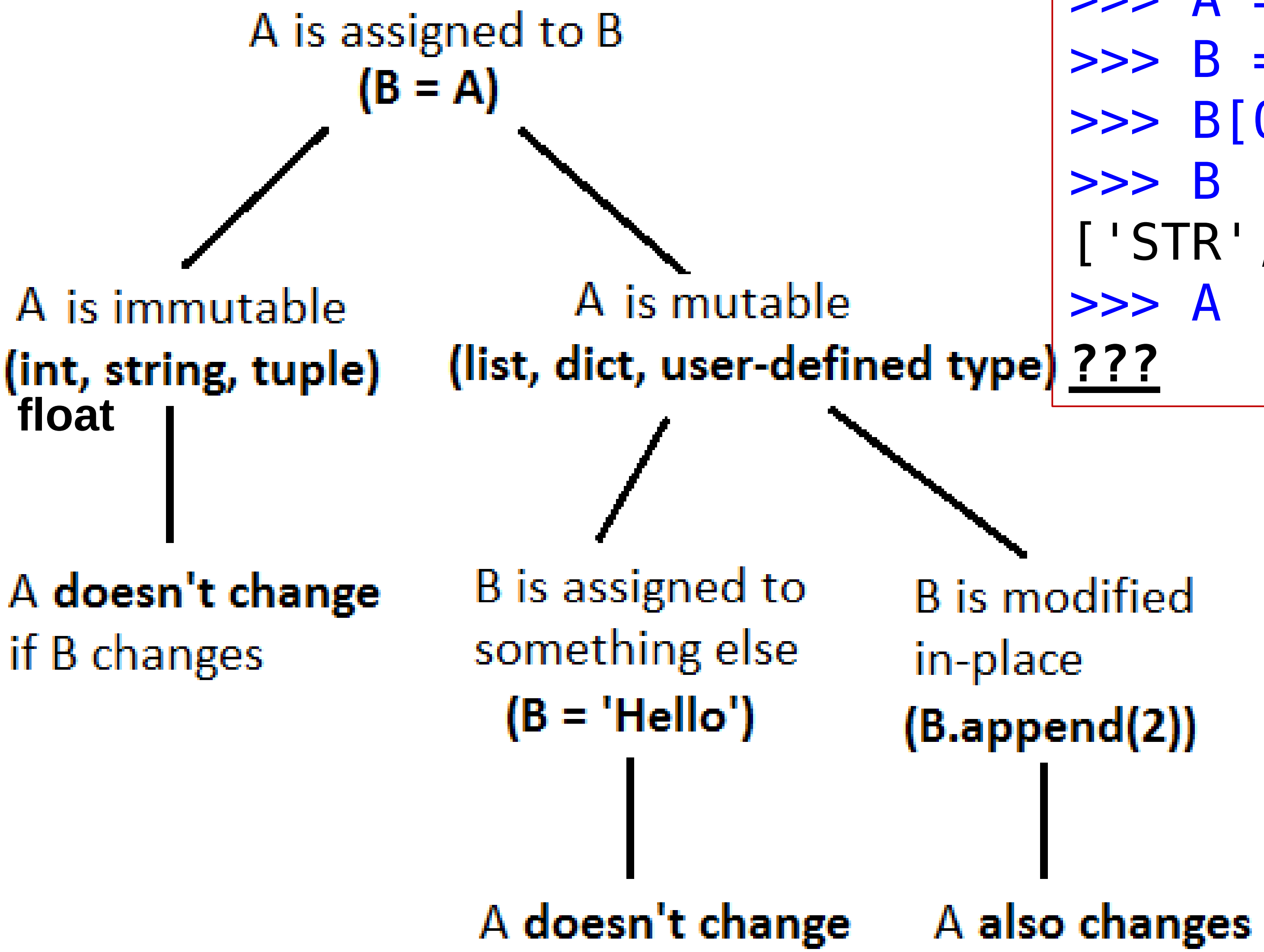
A **doesn't change**

B is modified in-place

(B.append(2))

A **also changes**

Functions that Modify Parameters



A is assigned to B

(B = A)

A is immutable

(int, string, tuple)

float

A **doesn't change**

if B changes

A is mutable

(list, dict, user-defined type)

B is assigned to something else

(B = 'Hello')

A **doesn't change**

B is modified in-place

(B.append(2))

A **also changes**

```
>>> A = [1, 2]
>>> B = A
>>> B[0] = 'STR'
>>> B
['STR', 2]
>>> A
???
```

Ned Batchelder - Facts and Myths about Python Names and Values - PyCon 2015

PyCon 2015

Montréal • April 8-16

Facts and Myths about Python names and values

Ned Batchelder

(23)

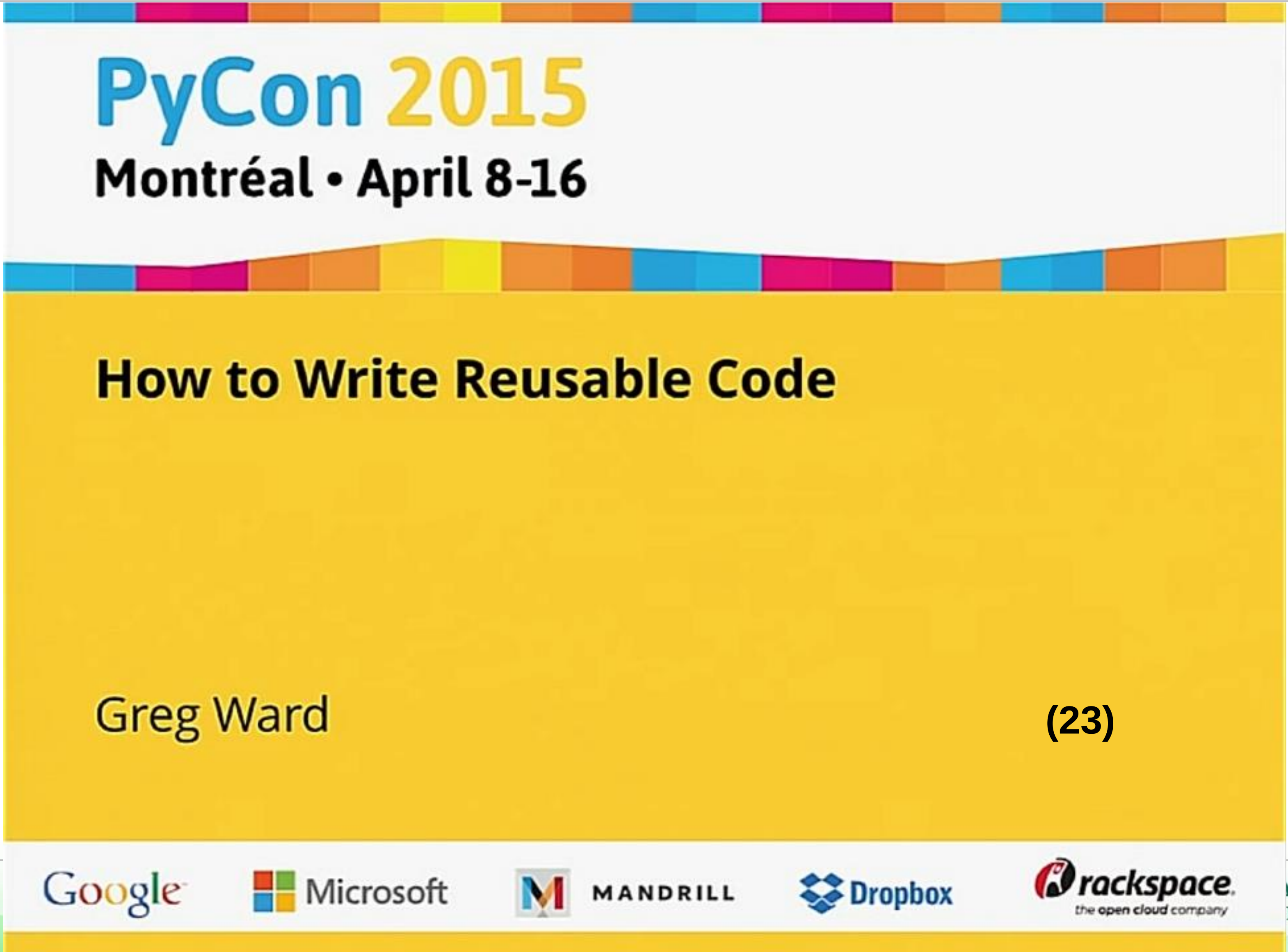
Functions and Program Structure

- In addition to reducing code duplication, using functions can also make your programs more *modular*

Functions and Program Structure

- As the program size increases, it gets more and more difficult to make sense out of it
- To deal with this complexity, you can break it down into smaller subprograms, each of which makes sense on its own
- Python libraries contain many reusable functions

Greg Ward - How to Write Reusable Code - PyCon 2015



PyCon 2015

Montréal • April 8-16

How to Write Reusable Code

Greg Ward

(23)

Google Microsoft MANDRILL Dropbox rackspace the open cloud company