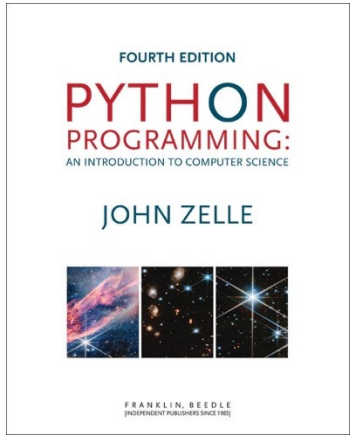
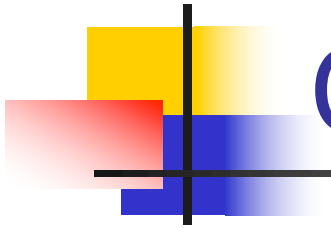
Python Programming: An Introduction to Computer Science

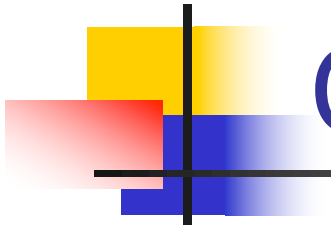


Chapter 5
Defining Functions



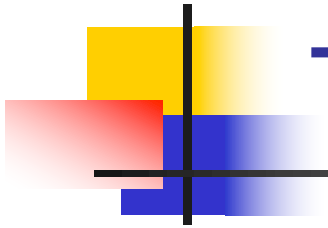
Objectives

- 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.



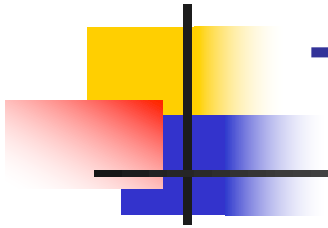
Objectives

- To write programs that use functions to reduce code duplication and increase program modularity.



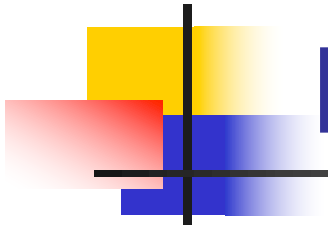
The Function of Functions

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



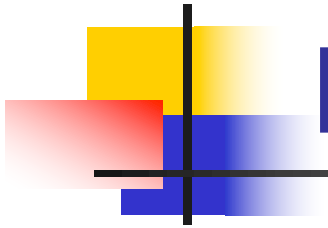
The Function of Functions

- Having similar or identical code in more than one place has some drawbacks.
  - 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.



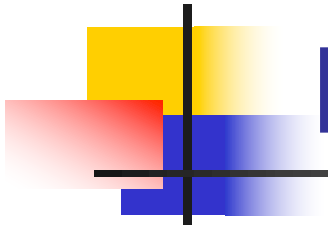
Functions, Informally

- 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 that 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*.



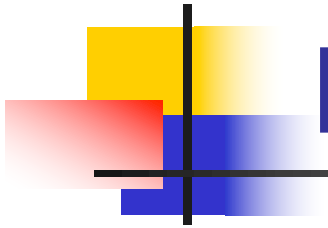
Functions, Informally

- 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!
```

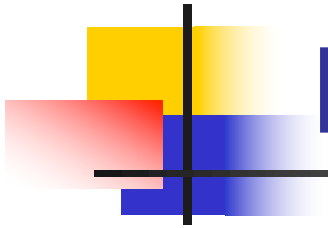


Functions, Informally

- There's some duplicated code in the program! (`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

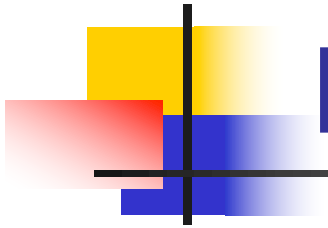
- The new program –

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

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

- Gives us this output –

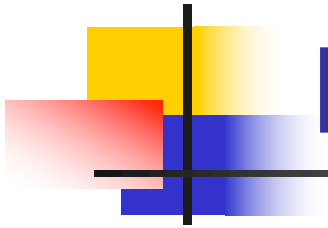
```
>>> singFred()
Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Fred...
Happy birthday to you!
```



Functions, Informally

- Creating this function saved us a lot of typing!
- 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

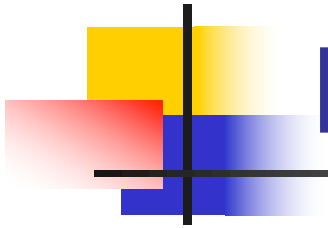
- We could write a main program to sing to both Lucy and Fred

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

- This gives us this new output

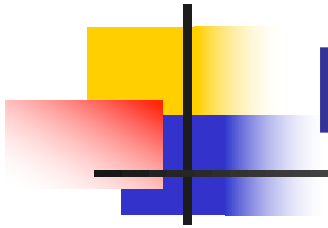
```
>>> main()
Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Fred..
Happy birthday to you!

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



Functions, Informally

- This is working great! But… there's still a lot of code duplication.
- The only difference between `singFred` and `singLucy` is the name in the third `print` statement.
- These two routines could be collapsed together by using a *parameter*.

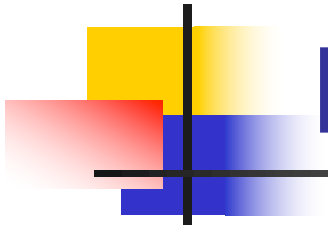


Functions, Informally

- The generic function *sing*

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

- This function uses a parameter named `person`. A *parameter* is a variable that is initialized when the function is called.

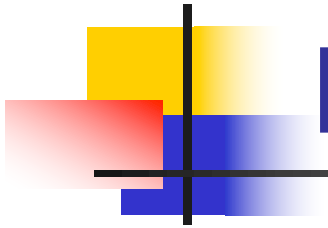


Functions, Informally

- Our new output –

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

- We can put together a new main program!



Functions, Informally

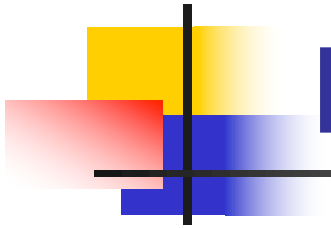
- Our new main program:

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

- Gives us this output:

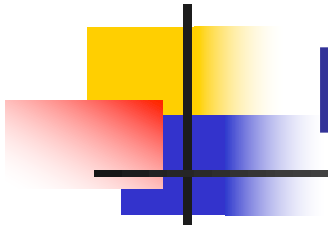
```
>>> main()
Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Fred.
Happy birthday to you!

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



Functions and Parameters: The Details

- Each function is its own little subprogram. The variables used inside of a function are *local* to that function, even if they happen to have the same name as 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.

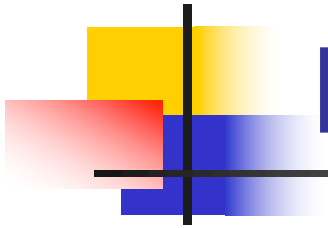


Functions and Parameters: The Details

- A function definition looks like this:

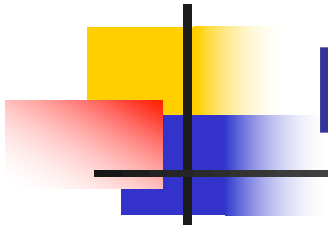
```
def <name>(<formal-parameters>):
    <body>
```

- The name of the function must be an identifier
- Formal-parameters is a (possibly empty) list of variable names



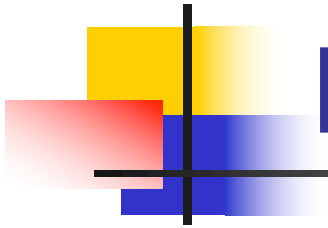
Functions and Parameters: The Details

- Formal parameters, like all variables used in the function, are only accessible in the body of the function. Variables with identical names elsewhere in the program are distinct from the formal parameters and variables inside of the function body.



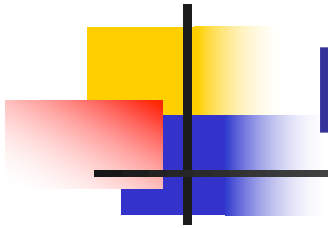
Functions and Parameters: The Details

- A function is called by using its name followed by a list of *actual parameters* or *arguments*.
  `<name>(<actual-parameters>)`
- When Python comes to a function call, it initiates a four-step process.



Functions and Parameters: The Details

- The function calling 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.

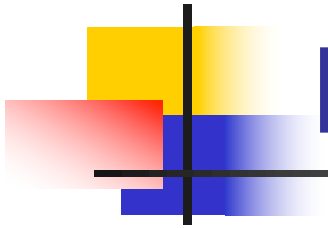


Functions and Parameters: The Details

- Let's trace through the following code:

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

- When Python gets to `sing("Fred")`, execution of `main` is temporarily suspended.
- Python looks up the definition of `sing` and sees that it has one formal parameter, `person`.



Functions and Parameters: The Detail

- The formal parameter is assigned the value of the actual parameter. It's as if the following statement had been executed:

```
person = "Fred"
```

Functions and Parameters: The Details

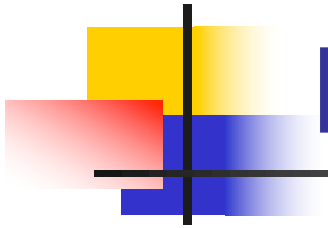
```
def main():
    sing("Fred")
    print()
    sing("Lucy")
```

person = "Fred"

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

person: "Fred"

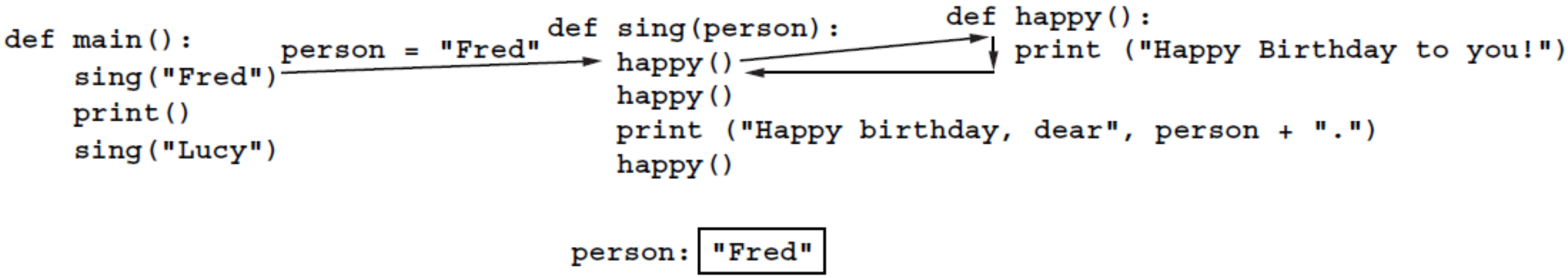
Note that the variable `person` has just been initialized.



Functions and Parameters: The Details

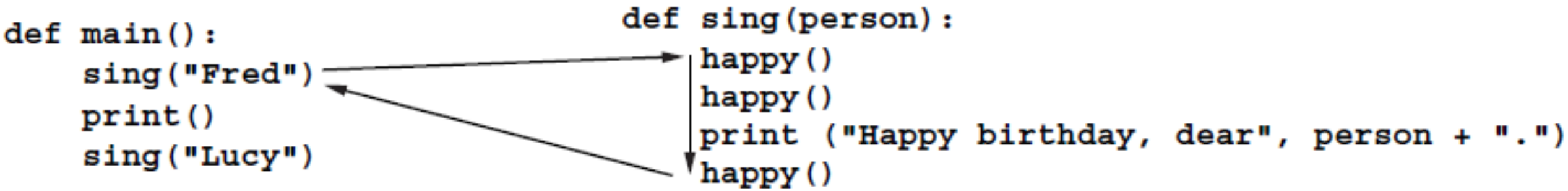
- At this point, Python begins executing the body of `sing`.
- The first statement is another function call, to `happy`. What happens next?
- Python suspends the execution of `sing` and transfers control to `happy`.
- `happy` consists of a single `print`, which is executed and control returns to where it left off in `sing`.

Functions and Parameters: The Details

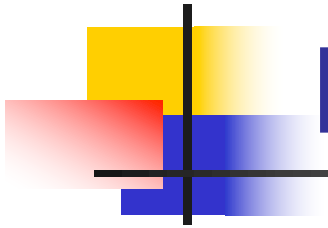


- Execution continues in this way with two more trips to `happy`.
- When Python gets to the end of `sing`, control returns to `main` and continues immediately following the function call.

Functions and Parameters: The Details



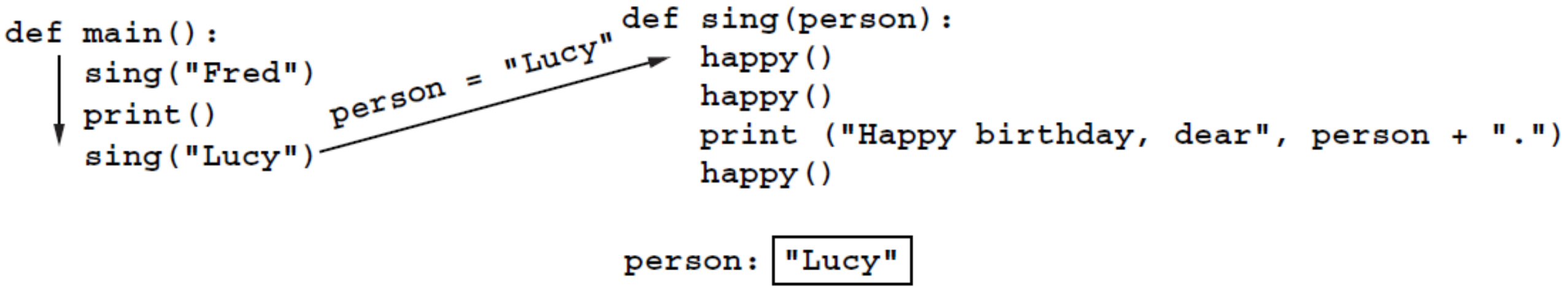
- Notice that the `person` variable in `sing` has disappeared!
- The memory occupied by local function variables is reclaimed when the function exits.
- Local variables do **not** retain any values from one function execution to the next.



Functions and Parameters: The Details

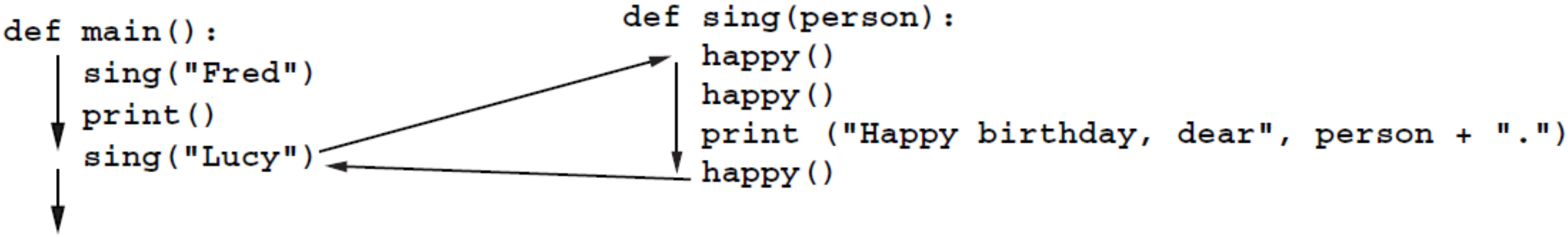
- The next statement is the bare `print`, which produces a blank line.
- Python encounters another call to `sing`, and control transfers to the `sing` function, with the formal parameter "Lucy".

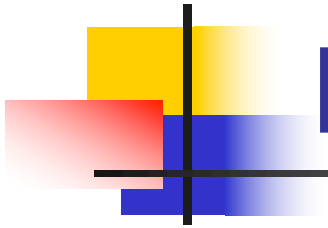
Functions and Parameters: The Details



- The body of `sing` is executed for Lucy with its three side trips to `happy` and control returns to `main`.

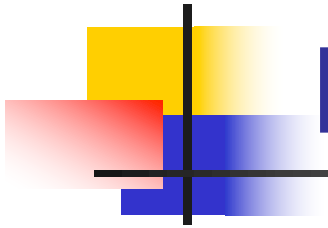
Functions and Parameters: The Details





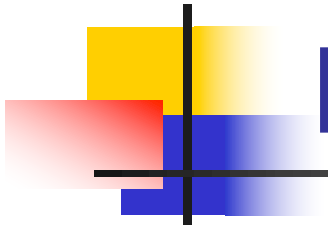
Functions and Paramters: The Details

- One thing not addressed in this example was multiple parameters. In this case the formal and actual parameters are matched up based on *position*, e.g. the first actual parameter is assigned to the first formal parameter, the second actual parameter is assigned to the second formal parameter, etc.



Functions That Return Values

- 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.

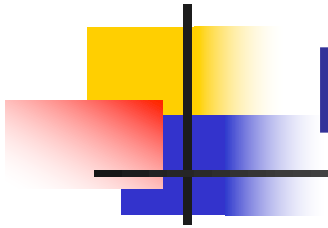


Functions That Return Values

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

```
square_root = math.sqrt(4)
```

- We say `sqrt` *returns* the square root of its argument.

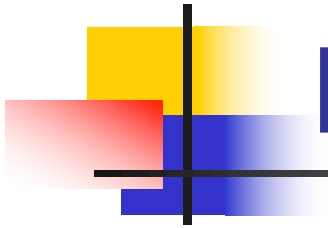


Functions That Return Values

- This function returns the square of a number:

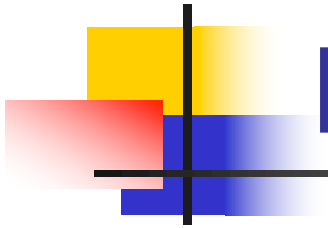
```
def square(x):
    return x*x
```

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



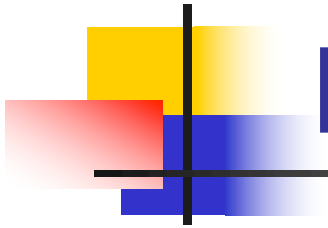
Functions That Return Values

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



Functions That Return Values

- The order of the function definitions in the program is not important.
- We just have to make sure a function is defined before the program actually tries to run it.
- Since our call to `main` does not usually happen until the every last line of the module, all of the functions will be defined before the program starts running.

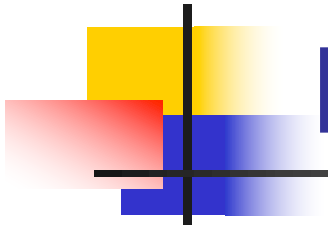


Functions That Return Values

- Through the magic of value-returning functions, we can streamline our program to a single string expression:

```
lyrics = (happy() + happy()
          + "Happy birthday, dear " + person + ".\n"
          + happy())
```

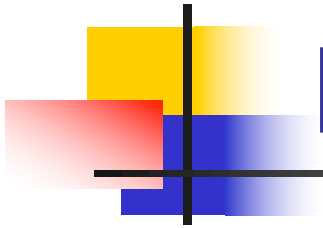
  - The right hand side is parenthesized so it can span several lines.
  - This makes use of string concatenation (+)
  - This demonstrates the power of value-returning functions.



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):
    plus = x + y
    minus = x - y
    return plus, minus
```

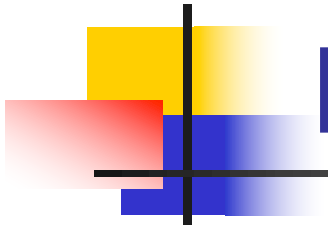


Functions That Return Values

- When calling this function, use unpacking to unpack the values from the returned tuple.

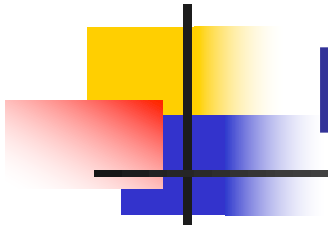
```
s, d = sumDiff(num1, num2)
print("The sum is", s, "and the difference is", d)
```

- As before, the values are assigned based on position, so *s* gets the first value returned (the sum), and *d* gets the second (the difference).



Functions That Return Values

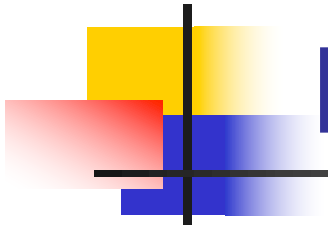
- One "gotcha" – all Python functions return a value, whether they contain a `return` statement or not. Functions without a `return` hand back a special object, denoted `None`.
- A common problem is writing a value-returning function and forgetting the `return`!



Functions That Return Values

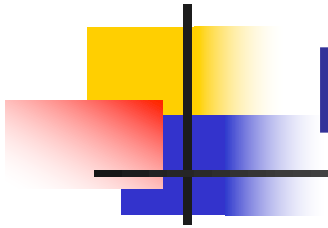
- If your value-returning functions produce strange messages, check to make sure you remembered to include the `return`!

```
def distance(p1, p2):
    dist = math.sqrt(square(p2.getX() - p1.getX())
                     + square(p2.getY() - p1.getY()))
perim = distance(p1, p2) + distance(p2, p3) + distance(p3, p1)
Traceback (most recent call last):
  File "<pyshell#24>", line 1, in <module>
    perim = distance(p1, p2) + distance(p2, p3) + distance(p3, p1)
TypeError: unsupported operand type(s) for +: 'NoneType' and 'NoneType'
```



Functions that Modify Parameters

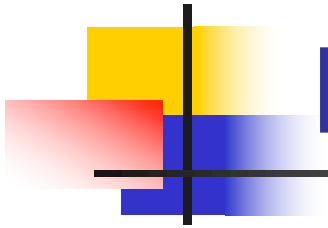
- Return values are the main way to send information from a function back to the caller.
- Sometimes, we can communicate back to the caller by making changes to the function parameters.
- Understanding when and how this is possible requires the mastery of some subtle details about how assignment works and the relationship between actual and formal parameters.



Functions that Modify Parameters

- Suppose you are writing a program that manages bank accounts. One function we would need to do is to accumulate interest on the account. Let's look at a first-cut at the function.

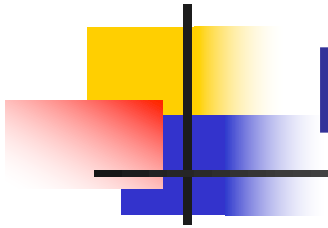
```
def addInterest(balance, rate):
    newBalance = balance * (1 + rate)
    balance = newBalance
```



Functions that Modify Parameters

- The intent is to set the balance of the account to a new value that includes the interest amount.
- Let's write a main program to test this:

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



Functions that Modify Parameters

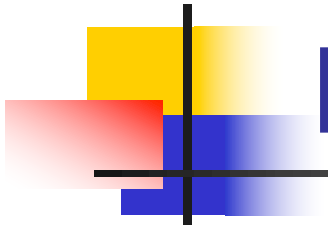
- We hope that that the 5% will be added to the amount, returning 1050.
- ```
  >>> test()
  1000
  ```
- What went wrong? Nothing!
  ```

Functions that Modify Parameters

- The first two lines of the test function create two local variables called `amount` and `rate` which are given the initial values of `1000` and `0.05`, respectively.

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

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

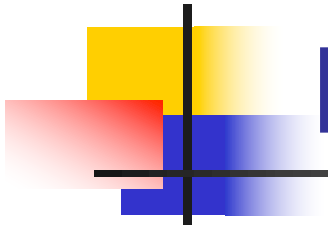


Functions that Modify Parameters

- Control then transfers to the `addInterest` function.
- The formal parameters `balance` and `rate` are assigned the values of the actual parameters `amount` and `rate`.
- Even though `rate` appears in both, they are separate variables (because of scope rules).

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

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



Functions that Modify Parameters

- The assignment of the parameters causes the variables `balance` and `rate` in `addInterest` to refer to the *values* of the actual parameters

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

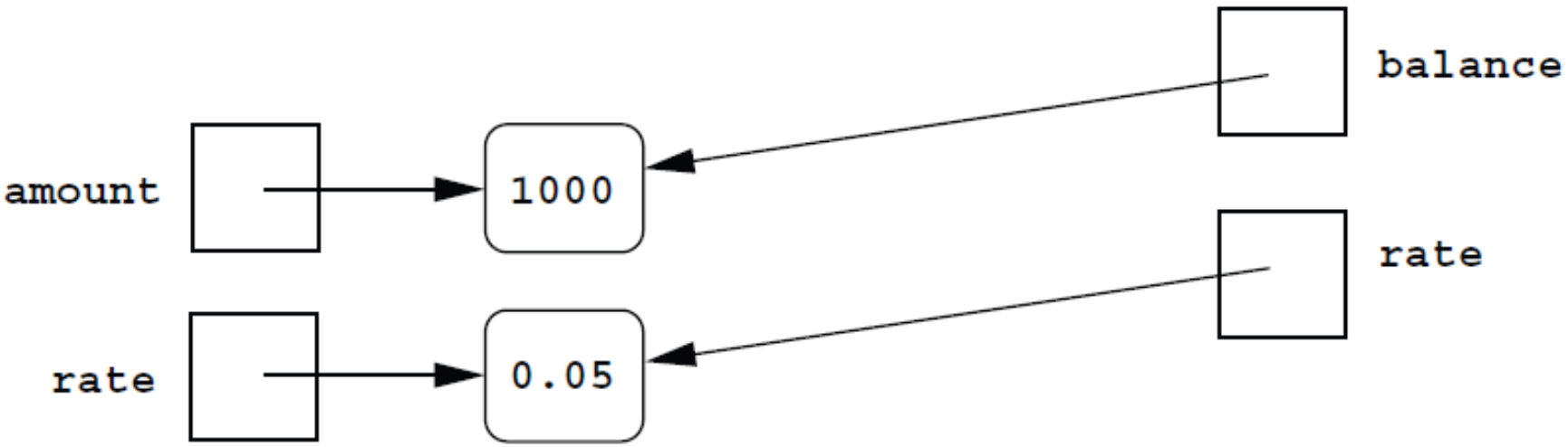
def test():
    amount = 1000
    rate = 0.05
    addInterest(amount, rate)
    print(amount)
```

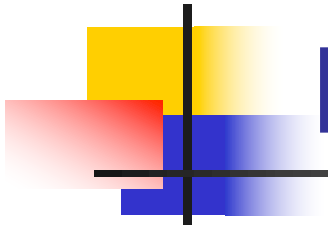
Functions that Modify Parameters

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

balance=amount
rate=rate

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





Functions that Modify Parameters

- Executing the first line of `addInterest` creates a new variable, `newBalance`.
- `balance` is then assigned the value of `newBalance`.

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

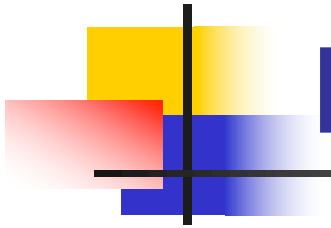
def test():
    amount = 1000
    rate = 0.05
    addInterest(amount, rate)
    print(amount)
```

Functions that Modify Parameters

- `balance` now refers to the same value as `newBalance`, but this had no effect on `amount` in the `test` function.

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

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



Functions that Modify Parameters

- `balance` now refers to the same value as `newBalance`, but this had no effect on `amount` in the `test` function.

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

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

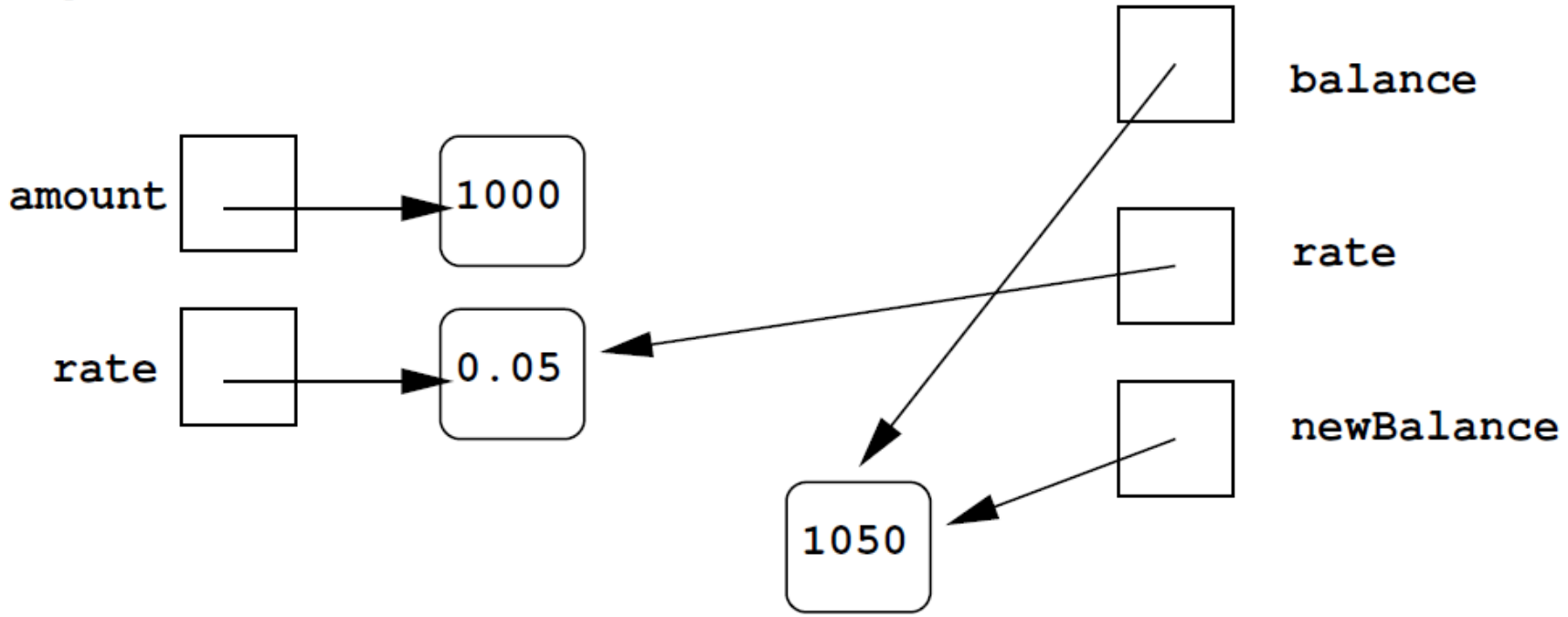
Functions that Modify Parameters

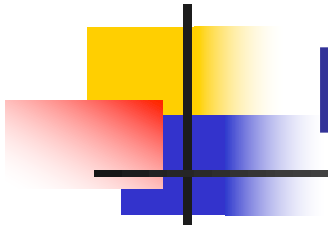
```
def test():
    amount = 1000
    rate = 0.05
    addInterest(amount,rate)
    print amount
```

balance=amount

rate=rate

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



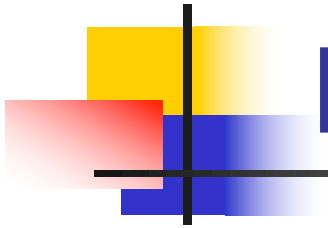


Functions that Modify Parameters

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

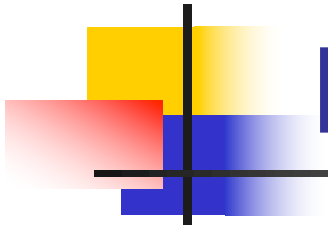
```
def addInterest(balance, rate):
    newBalance = balance * (1 + rate)
    balance = newBalance

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



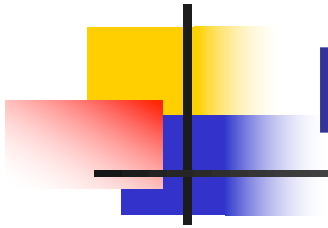
Functions that Modify Parameters

- 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.
- Python is said to pass all parameters *by value*.



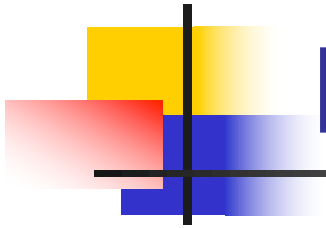
Functions that Modify Parameters

- Some programming languages (C++, Ada, and many more) do allow variables themselves to be sent as parameters to a function. This mechanism is said to pass parameters *by reference*.
- When a new value is assigned to the formal parameter, 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()
```