

CMSC201

Computer Science I for Majors

Lecture 11 – Functions (cont)

Last Class We Covered

- Functions
 - Why they're useful
 - When you should use them
- Defining functions
- Calling functions
- Variable scope
- Passing arguments

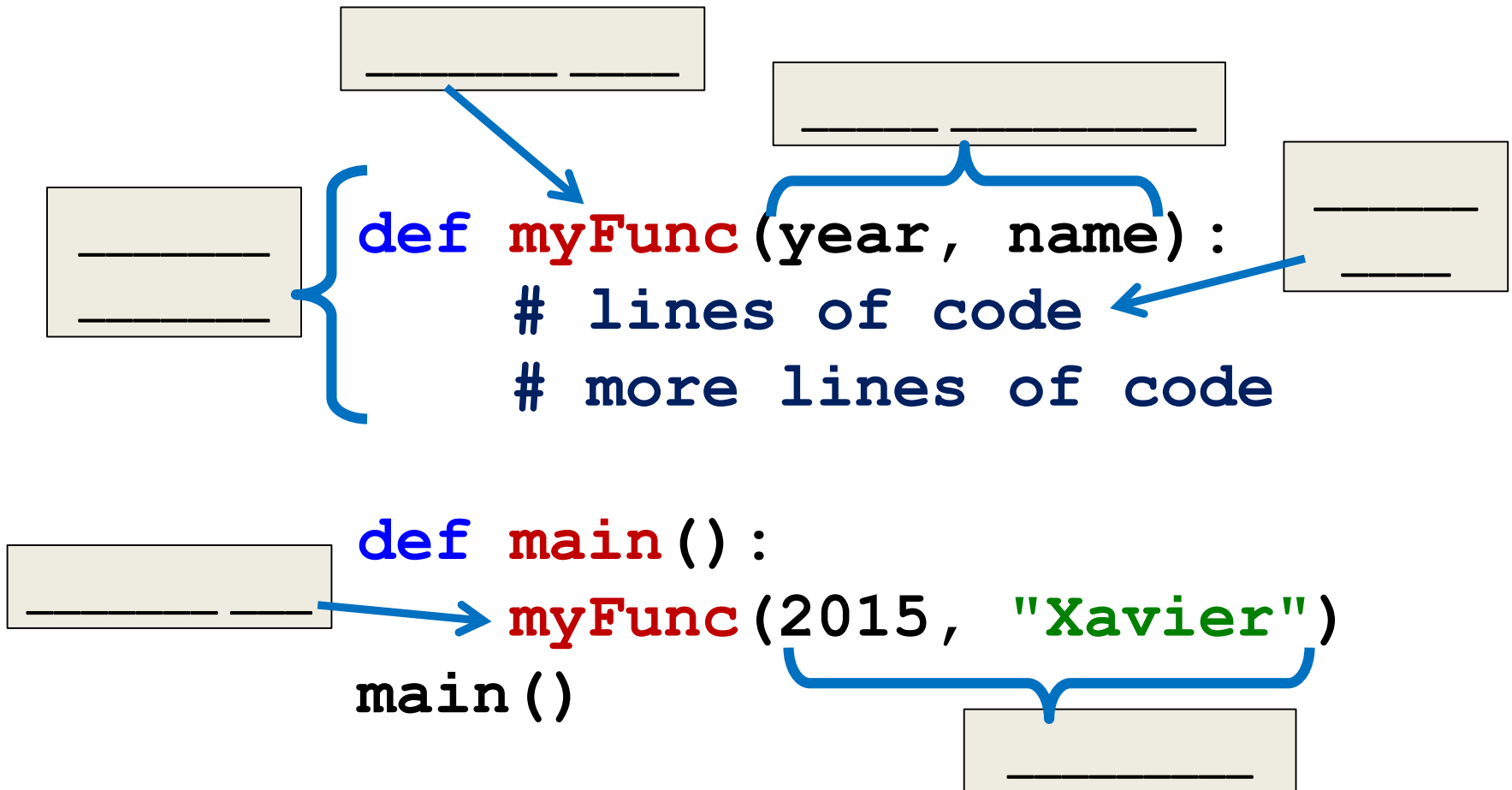
Any Questions from Last Time?

Today's Objectives

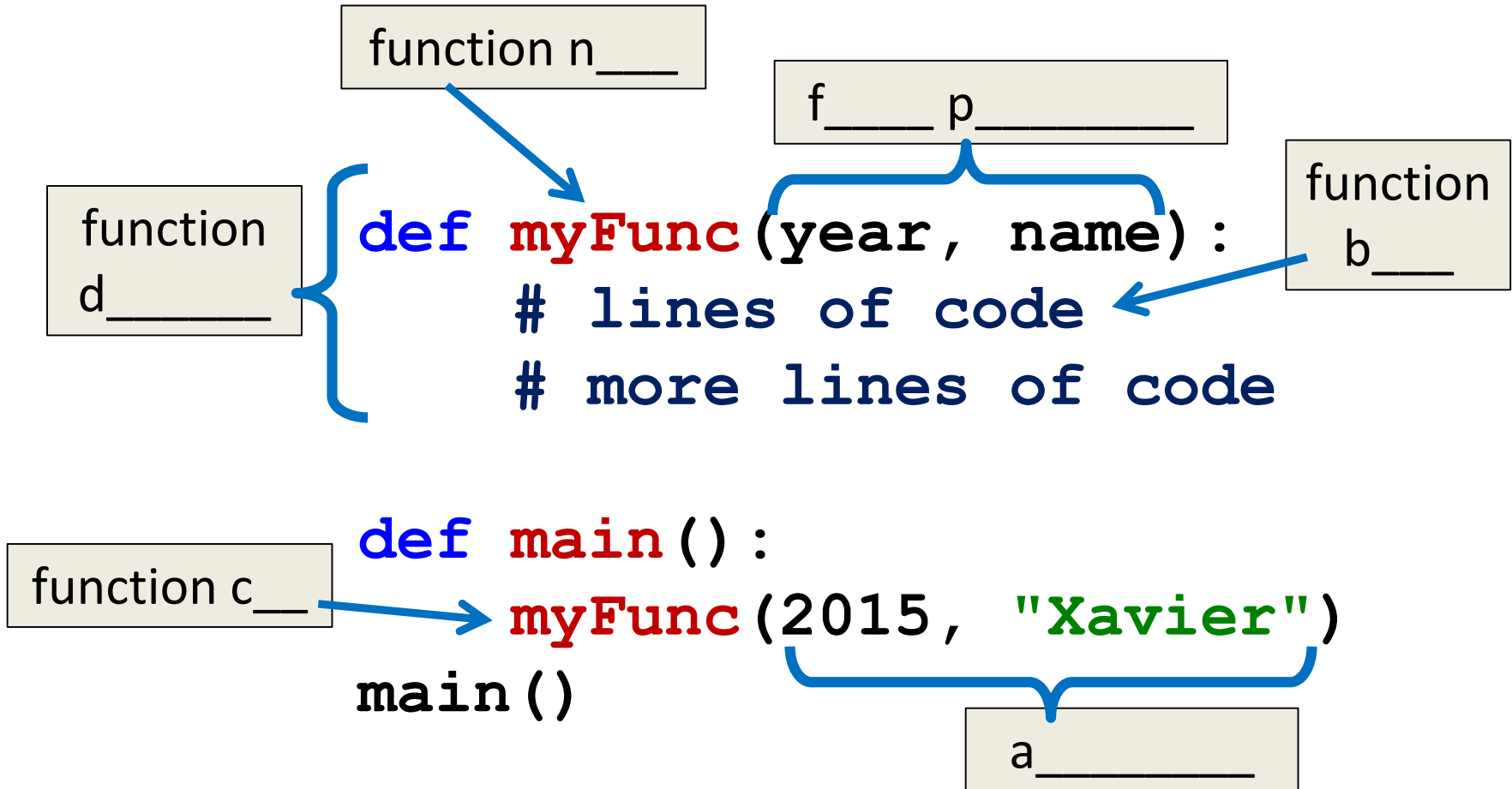
- To introduce value-returning functions
 - Common problems
 - Solutions to common problems
- To better grasp how values in the scope of a function actually work
- To practice function calls

Review: Parts of a Function

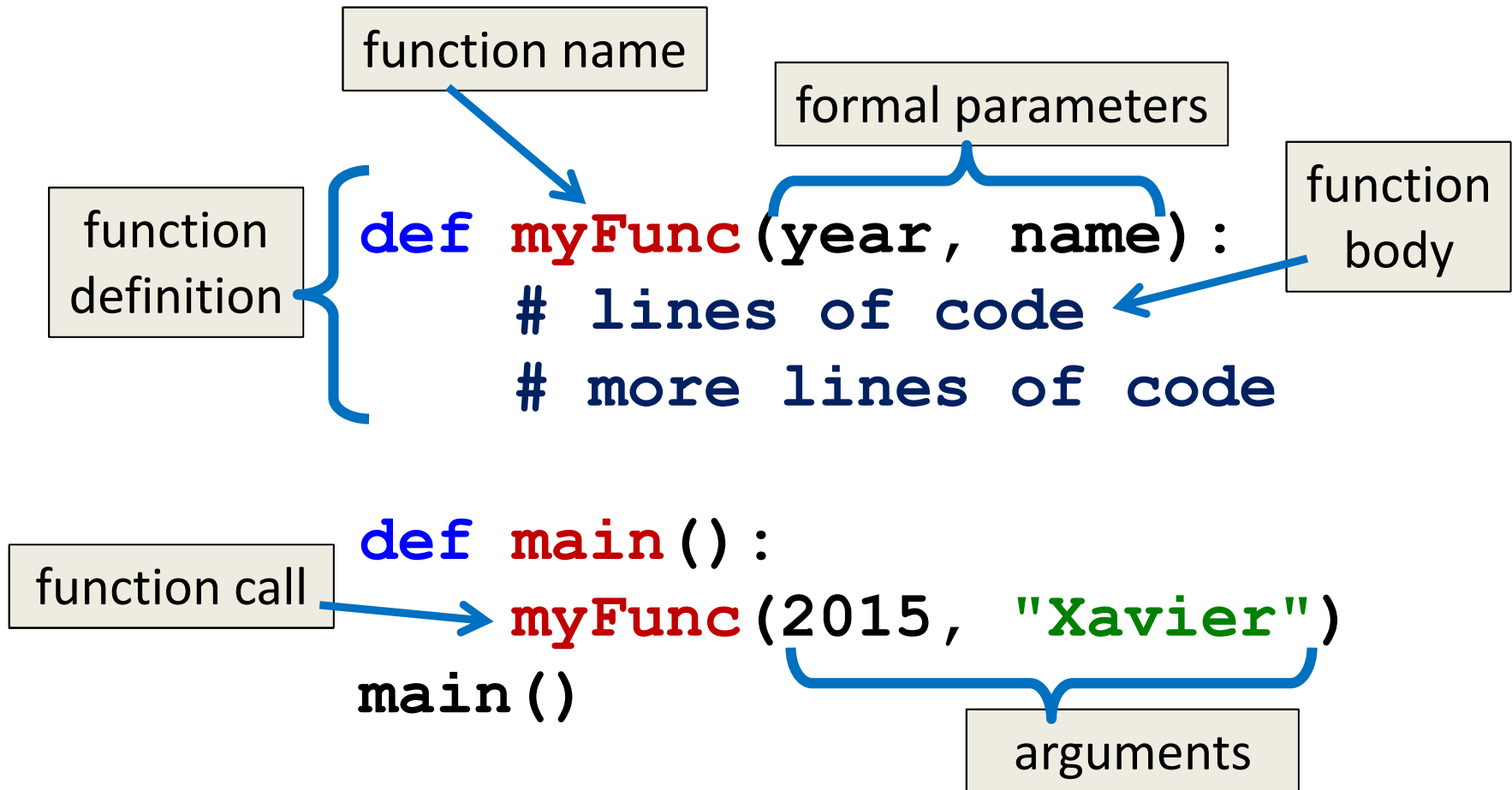
Function Vocabulary



Function Vocabulary

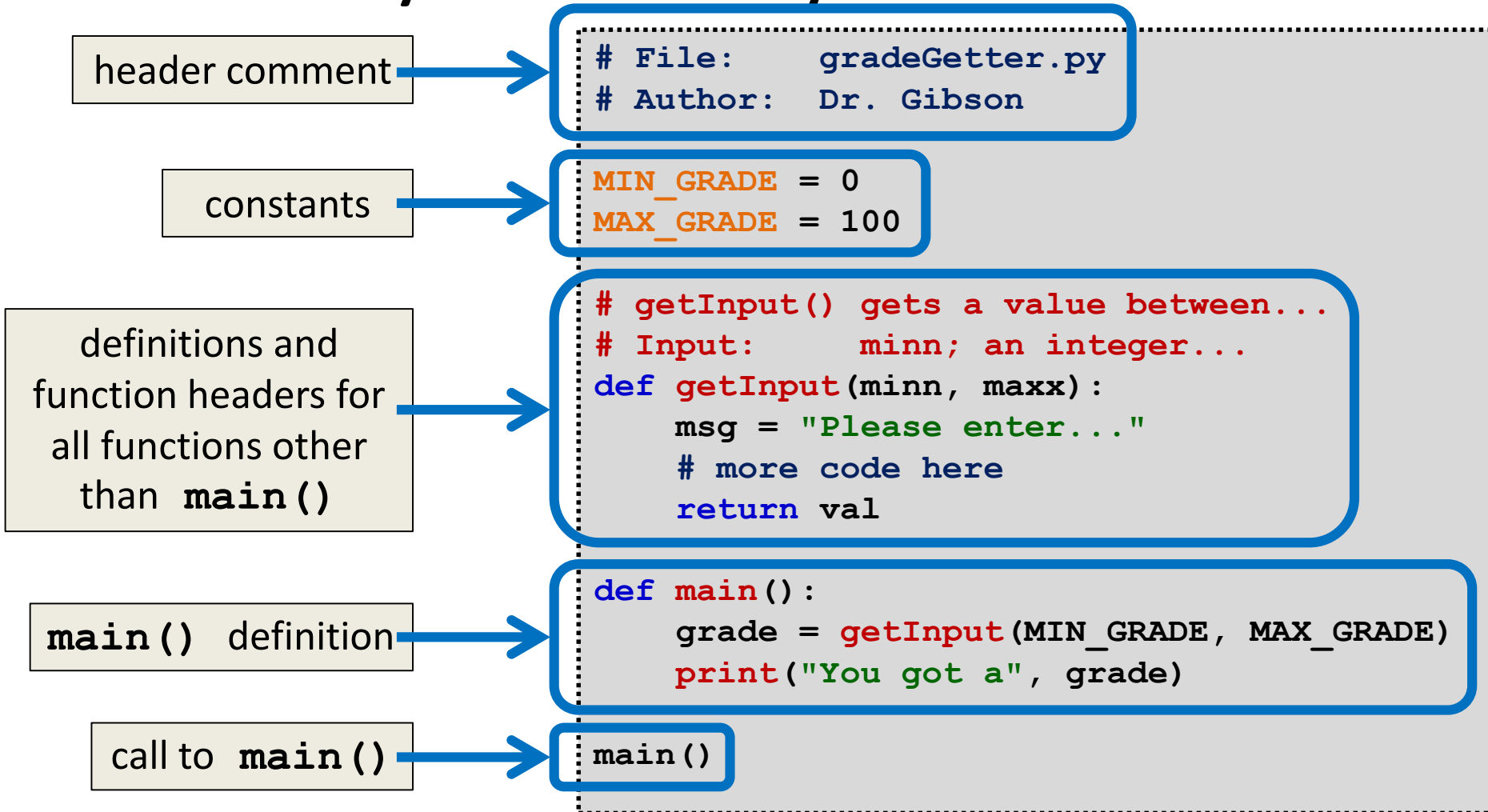


Function Vocabulary



File Layout and Constants

Layout of a Python File



Note On Global Constants

- ***Globals*** are variables declared outside of any function (including `main()`)
- Accessible globally in your program
 - To all functions and code
- Your programs may not have global variables
- Your programs may use global **constants**
 - In fact, constants should be global

Return Statements

Giving Information to a Function

- 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

Getting Information from a Function

- We've already seen numerous examples of functions that return values

`int()`, `len()`, `input()`, etc.

- For example, `len()`
 - Takes in any list or string as its parameter
 - Counts the number of elements (or characters)
 - And returns an integer value

Functions that Return Values

- To have a function return a value after it is called, we need to use the **return** keyword

```
def square (num) :  
    ans = num * num  
    # return the square  
    return ans
```

Handling Return Values

- When Python encounters **return**, it...
 - Exits the function (immediately!)
 - Even if it's not the end of the function
 - Returns control back to where the function was called from
- The expression in the return statement is evaluated, then sent back to the caller as a ***return value***

Code Trace: Return from `square()`

Let's follow the flow of the code

```
def main():  
    x = 5  
    y = square(x)  
    print(y)
```

→ `main()`

Step 1: Call `main()`

```
def square(num):  
    ans = num * num  
    return ans
```

Code Trace: Return from `square()`

Let's follow the flow of the code

```
def main():  
    x = 5  
    y = square(x)  
    print(y)
```

→ `main()`

Step 1: Call `main()`

Step 2: Pass control to `def main()`

```
def square(num):  
    ans = num * num  
    return ans
```

Code Trace: Return from `square()`

Let's follow the flow of the code

`x:` 5

→ `def main():`

`x = 5`

`y = square(x)`

`print(y)`

`main()`

`def square(num):`

`ans = num * num`

`return ans`

Step 1: Call `main()`

Step 2: Pass control to `def main()`

Step 3: Set `x = 5`

Code Trace: Return from `square()`

Let's follow the flow of the code

`x:` 5

```
def main():  
→ x = 5  
  y = square(x)  
  print(y)  
main()
```

```
def square(num):  
  ans = num * num  
  return ans
```

Step 1: Call `main()`

Step 2: Pass control to `def main()`

Step 3: Set `x = 5`

Step 4: See the function call to `square()`

Code Trace: Return from `square()`

Let's follow the flow of the code

```
def main():  
    x = 5  
    → y = square(x)  
    print(y)  
main()
```

x: 5

argument:
5

```
def square(num):  
    ans = num * num  
    return ans
```

Step 1: Call `main()`

Step 2: Pass control to `def main()`

Step 3: Set `x = 5`

Step 4: See the function call to `square()`

Step 5: Pass control from `main()` to `square()`, sending the argument 5

Code Trace: Return from `square()`

Let's follow the flow of the code

```
def main():  
    x = 5  
    y = square(x)  
    print(y)  
main()
```

x:

→ def square(num):
 ans = num * num
 return ans

num:

Step 1: Call `main()`

Step 2: Pass control to `def main()`

Step 3: Set `x = 5`

Step 4: See the function call to `square()`

Step 5: Pass control from `main()` to `square()`, sending the argument 5

Step 6: Set the value of the formal parameter `num` in `square()` to 5

Code Trace: Return from `square()`

Let's follow the flow of the code

```
def main():  
    x = 5  
    y = square(x)  
    print(y)  
main()
```

x:

→ def square(num):
 ans = num * num
 return ans

num:
ans:

Step 1: Call `main()`

Step 2: Pass control to `def main()`

Step 3: Set `x = 5`

Step 4: See the function call to `square()`

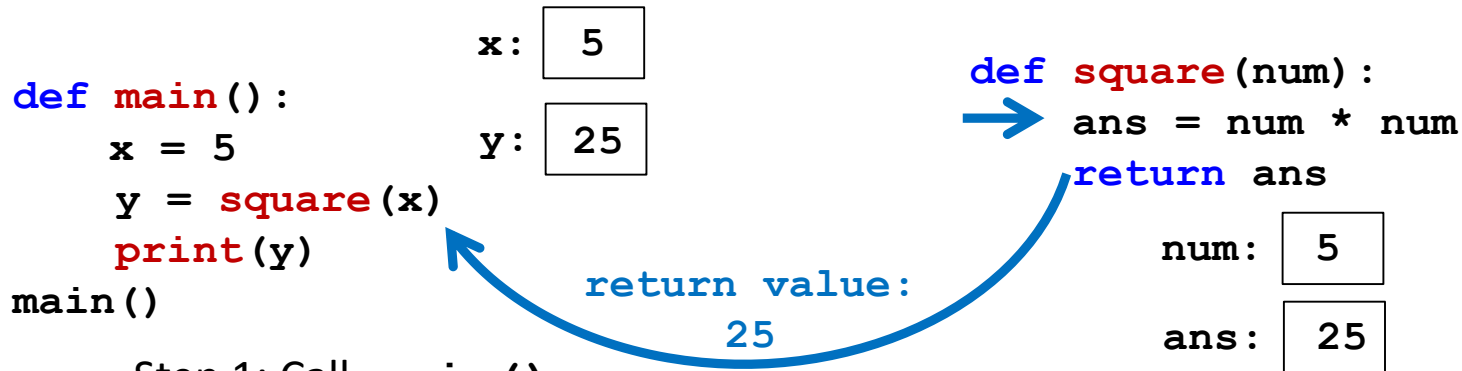
Step 5: Pass control from `main()` to `square()`, sending the argument 5

Step 6: Set the value of the formal parameter `num` in `square()` to 5

Step 7: Calculate `ans = num * num`

Code Trace: Return from `square()`

Let's follow the flow of the code



Step 1: Call `main()`

Step 2: Pass control to `def main()`

Step 3: Set `x = 5`

Step 4: See the function call to `square()`

Step 5: Pass control from `main()` to `square()`, sending the argument 5

Step 6: Set the value of the formal parameter `num` in `square()` to 5

Step 7: Calculate `ans = num * num`

Step 8: Return the value 25 to `main()` and set `y =` the returned value

Code Trace: Return from `square()`

Let's follow the flow of the code

```
def main():  
    x = 5  
    → y = square(x)  
    print(y)  
main()
```

x:
y:

```
def square(num):  
    ans = num * num  
    return ans
```

Step 1: Call `main()`

Step 2: Pass control to `def main()`

Step 3: Set `x = 5`

Step 4: See the function call to `square()`

Step 5: Pass control from `main()` to `square()`, sending the argument `5`

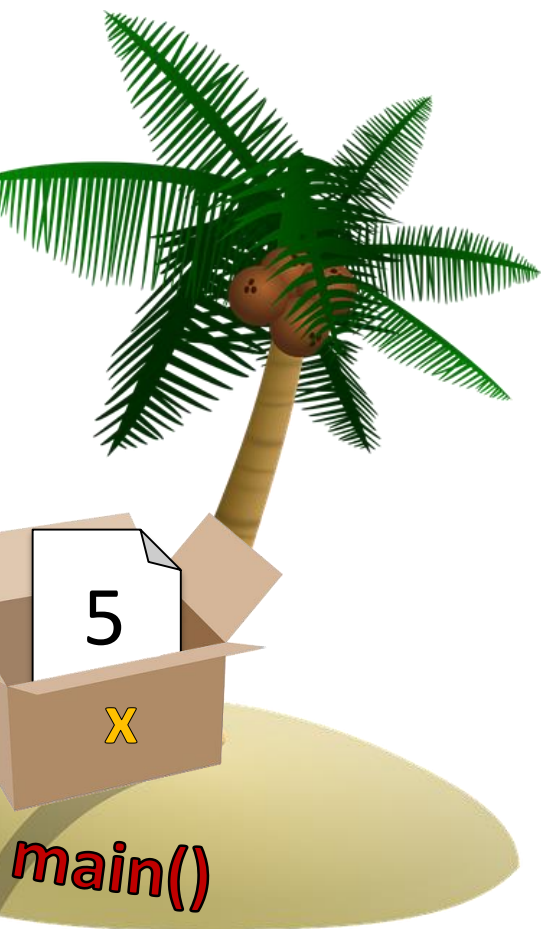
Step 6: Set the value of the formal parameter `num` in `square()` to `5`

Step 7: Calculate `ans = num * num`

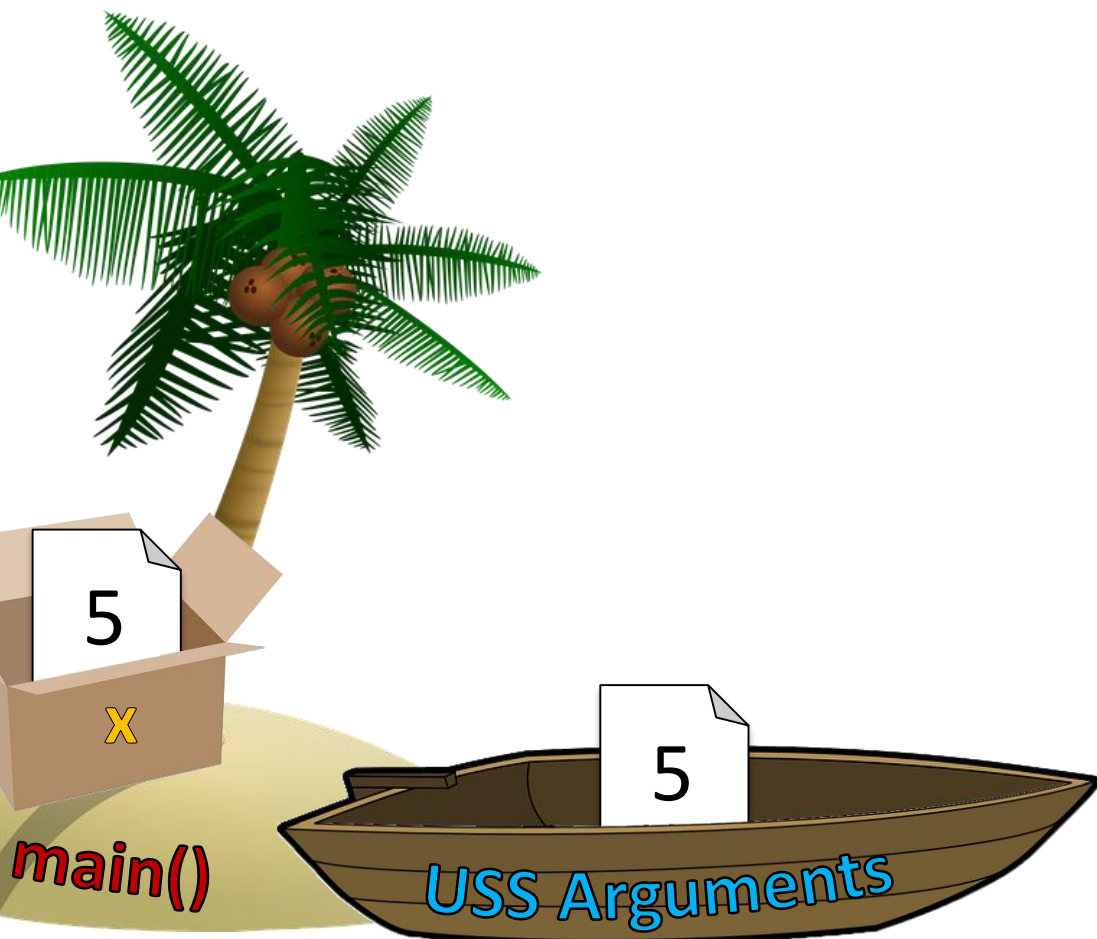
Step 8: Return the value `25` to `main()` and set `y =` the returned value

Step 9: Print value of `y`

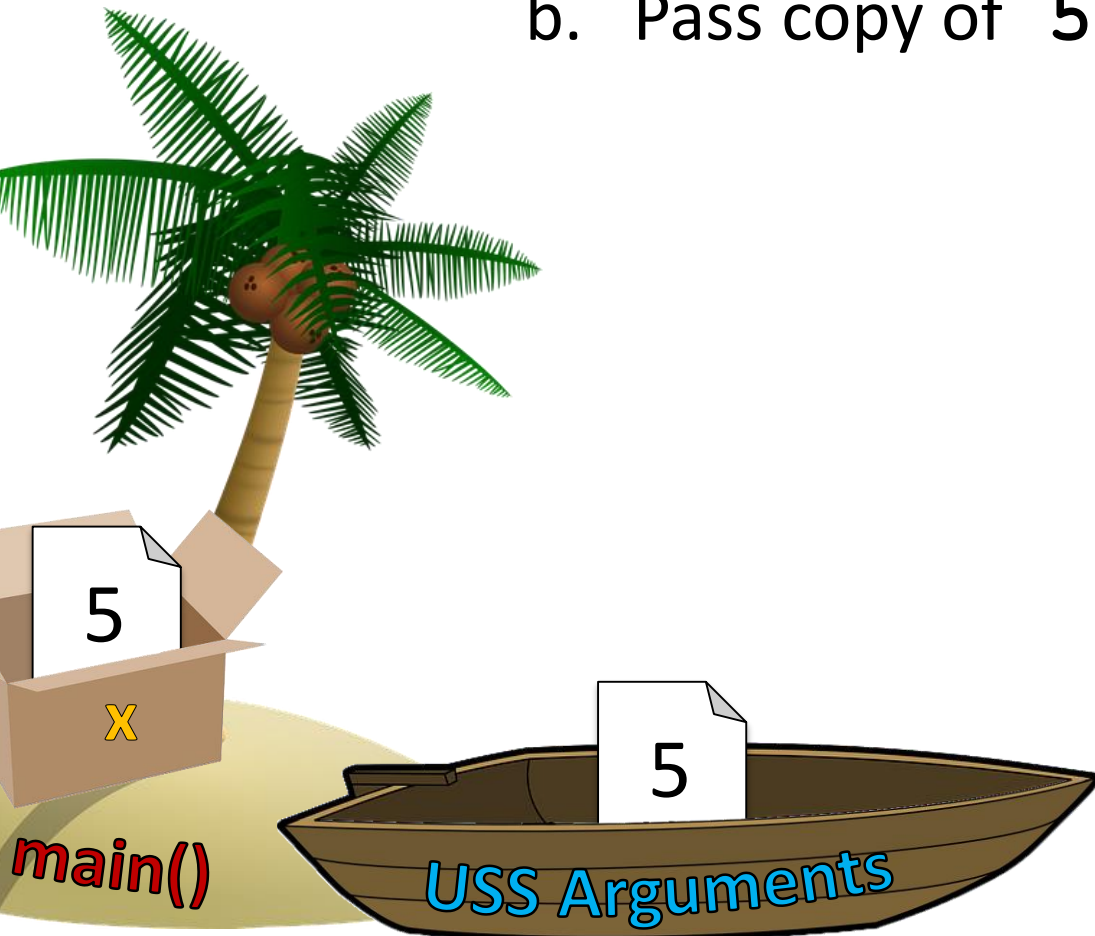
Island Example



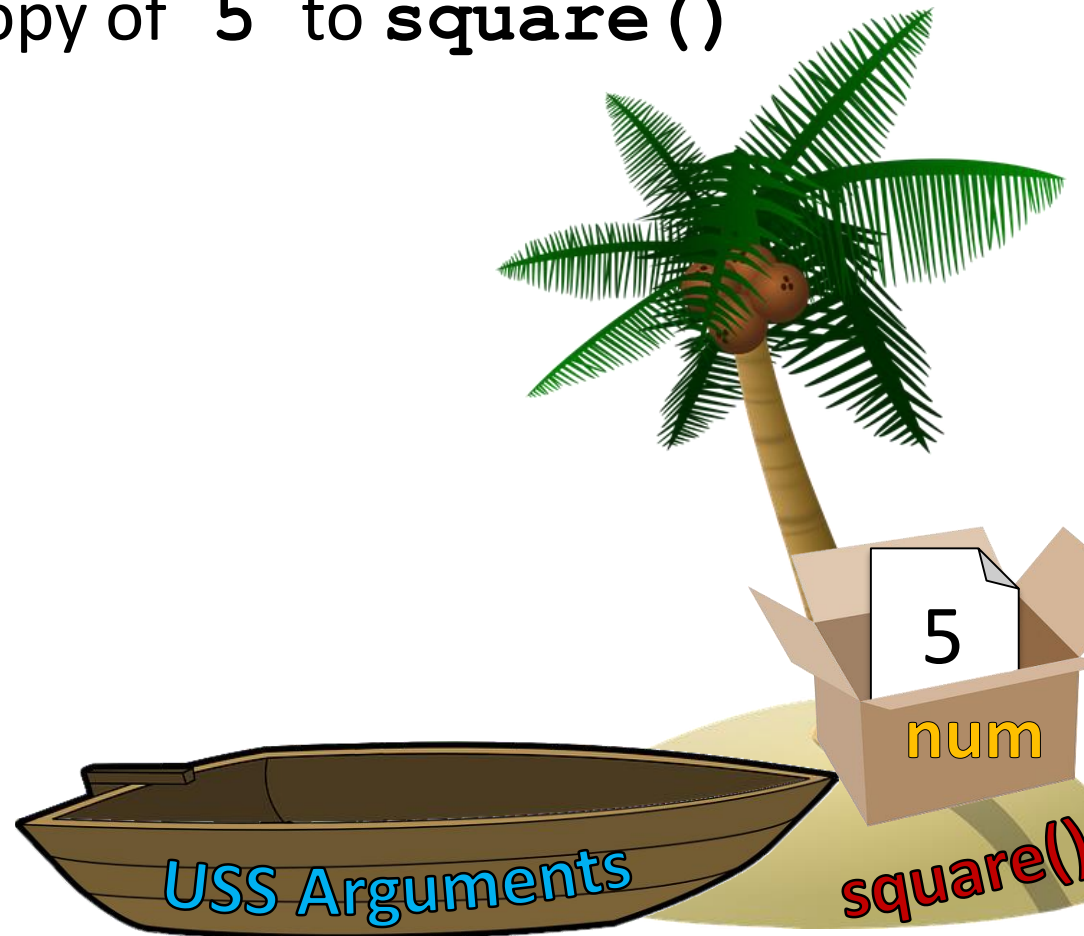
1. Function `square()` is called
 - a. Make copy of `x`'s value



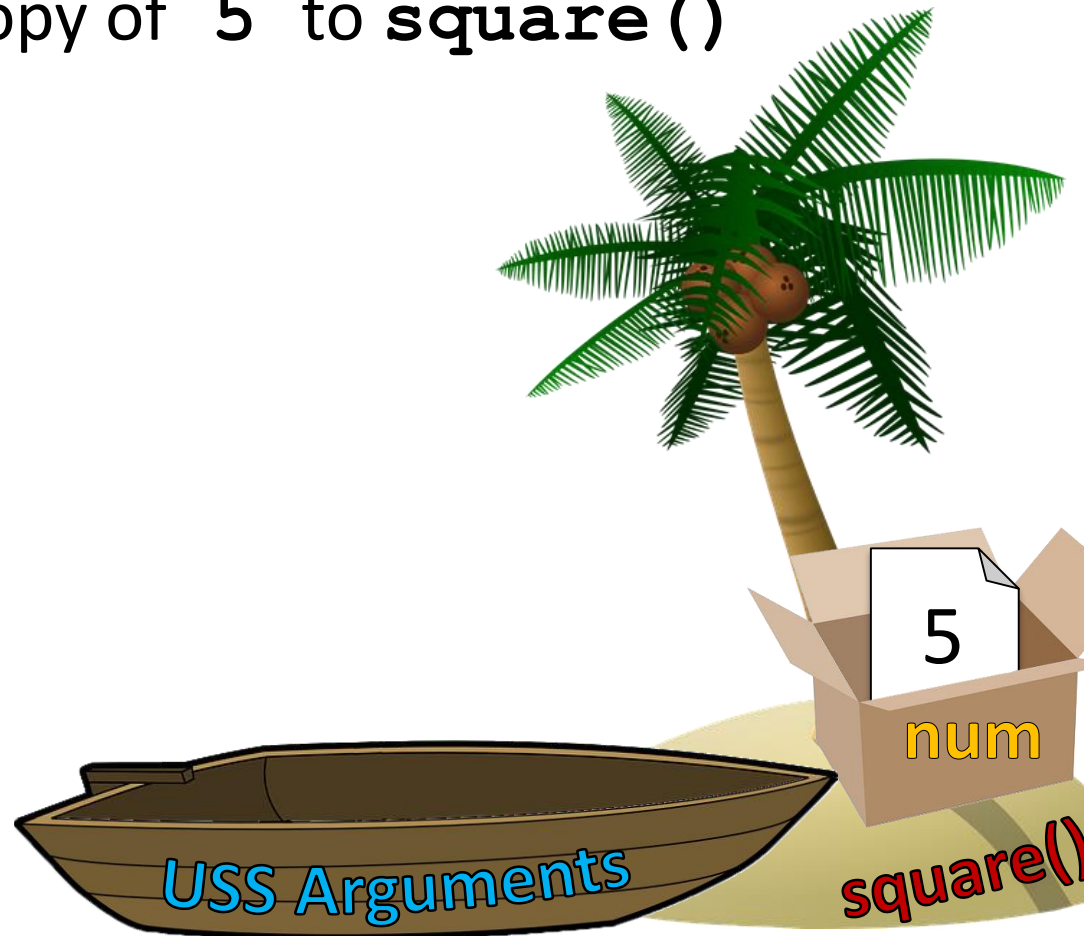
1. Function `square()` is called
 - a. Make copy of `x`'s value
 - b. Pass copy of `5` to `square()`



1. Function `square()` is called
 - a. Make copy of `x`'s value
 - b. Pass copy of `5` to `square()`



1. Function `square()` is called
 - a. Make copy of `x`'s value
 - b. Pass copy of `5` to `square()`

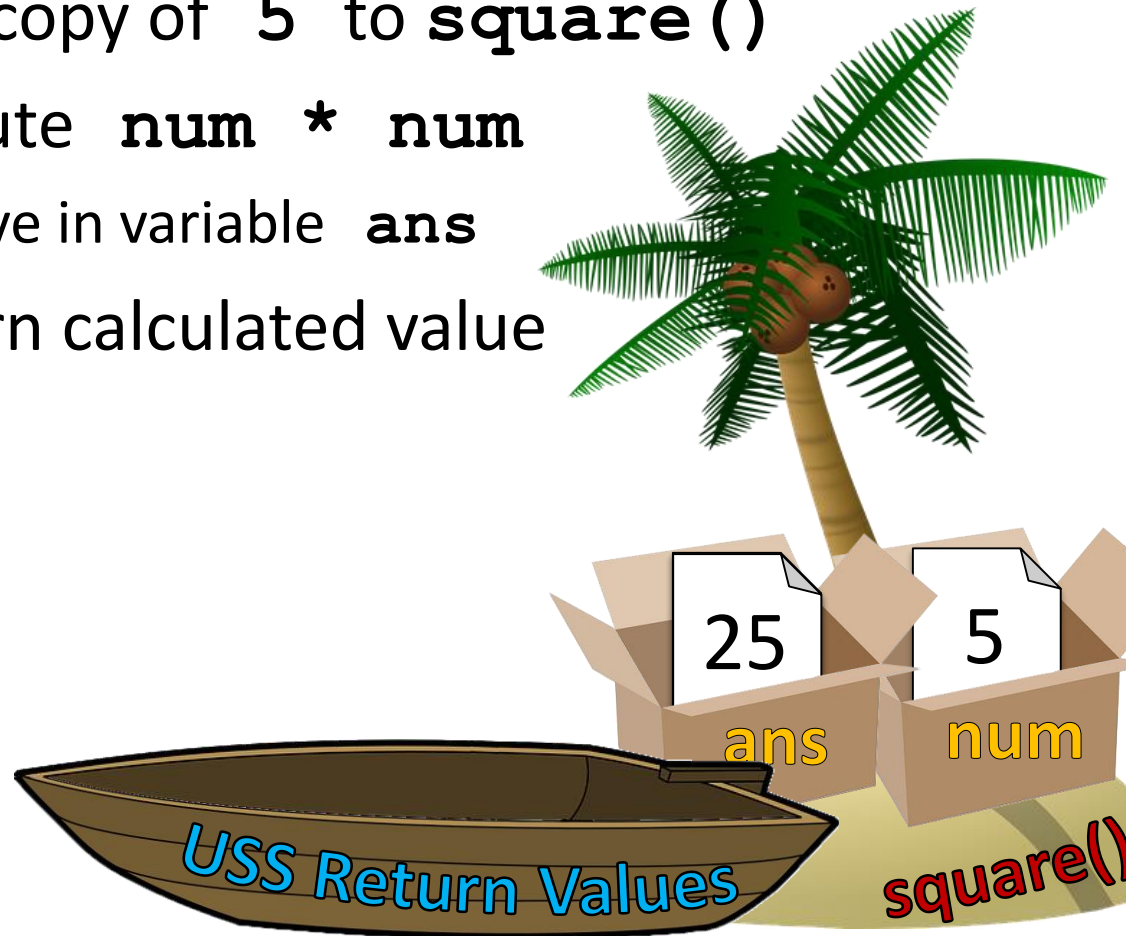


1. Function `square()` is called

- a. Make copy of `x`'s value
- b. Pass copy of `5` to `square()`
- c. Execute `num * num`
 - a. Save in variable `ans`

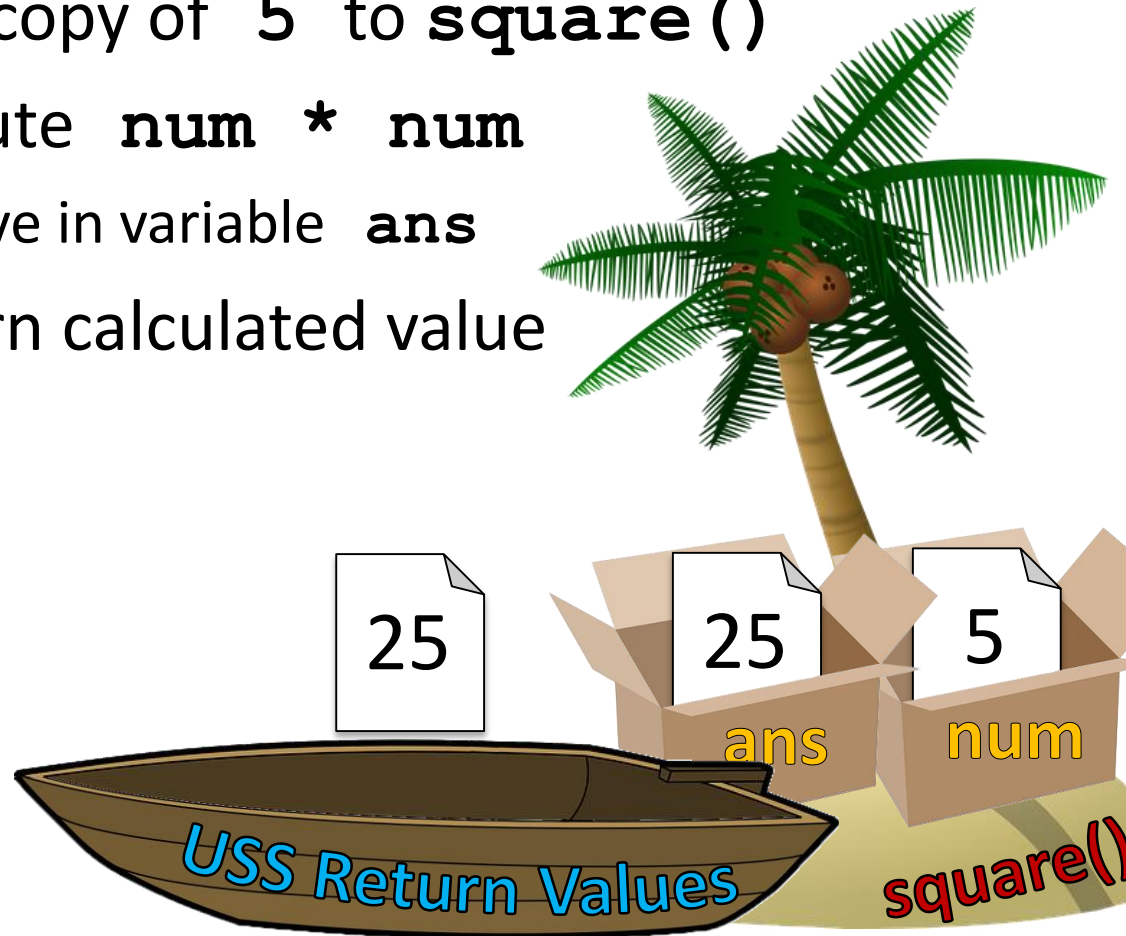


1. Function `square()` is called
 - a. Make copy of `x`'s value
 - b. Pass copy of `5` to `square()`
 - c. Execute `num * num`
 - a. Save in variable `ans`
 - d. Return calculated value



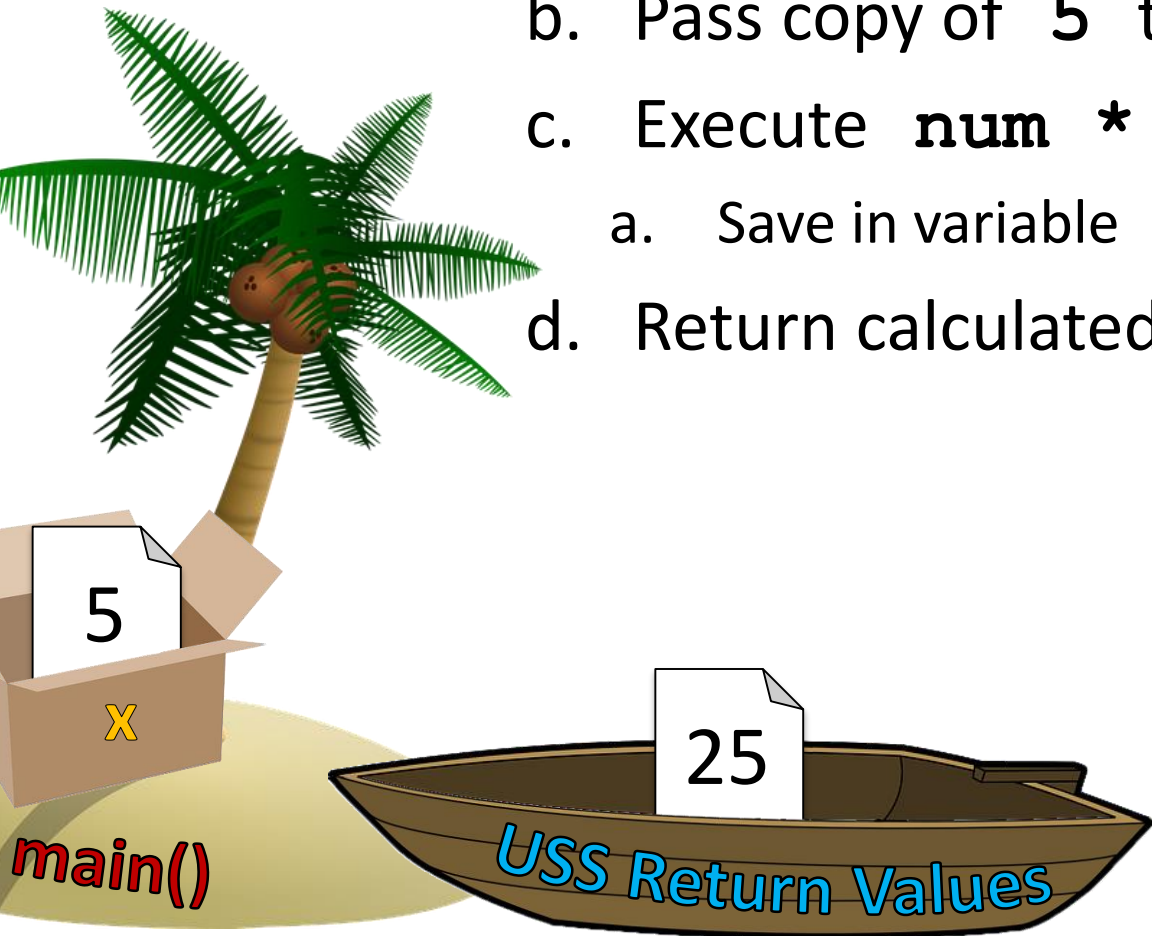
1. Function `square()` is called

- a. Make copy of `x`'s value
- b. Pass copy of `5` to `square()`
- c. Execute `num * num`
 - a. Save in variable `ans`
- d. Return calculated value



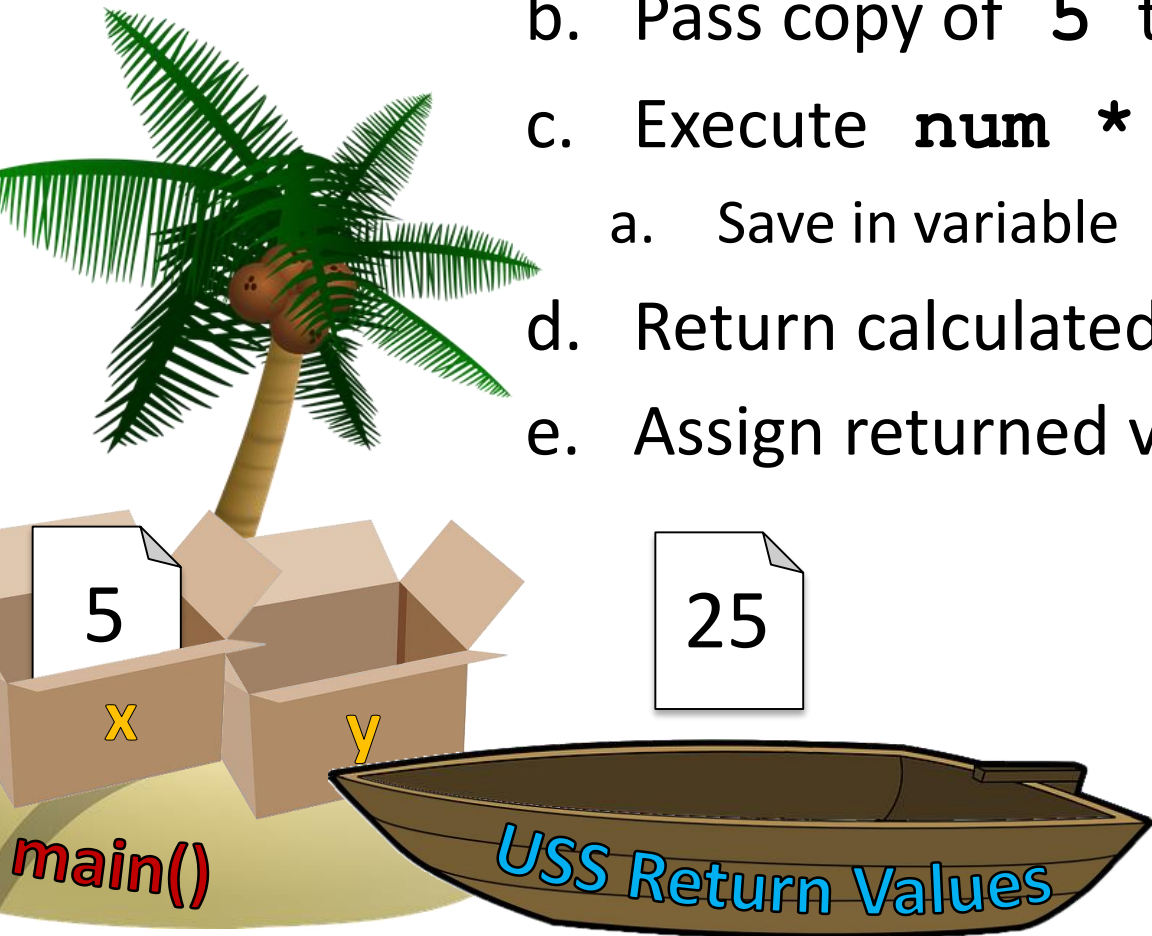
1. Function `square()` is called

- a. Make copy of `x`'s value
- b. Pass copy of `5` to `square()`
- c. Execute `num * num`
 - a. Save in variable `ans`
 - d. Return calculated value



1. Function `square ()` is called

- a. Make copy of `x`'s value
- b. Pass copy of `5` to `square ()`
- c. Execute `num * num`
 - a. Save in variable `ans`
- d. Return calculated value
- e. Assign returned value to `y`



None and Common Problems

Every Function Returns *Something*

- All Python functions return a value
 - Even if they don't have a **return** statement
- Functions without an explicit **return** pass back a special object, called **None**
 - **None** is the absence of a value

Example (That We'll Break Soon)

- Here is a simple toy example:

```
def multiply(num1, num2):  
    print("doing", num1, "*", num2)  
    answer = num1 * num2  
    return answer
```

What is the output
of this code?

- Assume that this code is in `main()`:

```
product = multiply(6, 3)  
print("result is", product)
```

```
doing 6 * 3  
result is 18
```

Problem #1

- Forgetting to write a `return` statement

```
def multiply(num1, num2):  
    print("doing", num1, "*", num2)  
    answer = num1 * num2
```

```
product = multiply(3, 5)  
print("result is", product)
```

- What is the code's output now?

Problem #1

- Forgetting to write a `return` statement

```
def multiply(num1, num2):  
    print("doing", num1, "*", num2)  
    answer = num1 * num2
```

```
product = multiply(3, 5)  
print("result is", product)
```

doing 3 * 5
result is None

Variable given the
return value has a
value of **None**

- What is the co

Problem #2

- Forgetting to assign the returned value

```
def multiply(num1, num2):  
    print("doing", num1, "*", num2)  
    return num1 * num2
```

```
multiply(7, 8)  
print("result is", product)
```


- What is the code's output now?

Problem #2

- Forgetting to assign the returned value

```
def multiply(num1, num2):  
    print("doing", num1, "*", num2)  
    return num1 * num2
```

```
multiply(7, 8)  
print("result is", product)
```



doing 7 * 8
[syntax error]

- What is the code's output?
Should have assigned `product` to the return value of `multiply`

Common Errors and Problems

- If your value-returning functions produce strange messages, check to make sure you used the **return** correctly!

```
TypeError: unsupported operand type(s)  
for *: 'NoneType' and 'int'
```

```
TypeError: 'NoneType' object is not  
iterable
```

“Modifying” Parameters

Bank Interest Example

- Suppose you are writing a program that manages bank accounts
- One function we would need to create is one to accumulate interest on the account

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

Bank Interest Example

- We want to set the balance of the account to a new value that includes the interest amount

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

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

What is the output of this code?

1000

Is this what we wanted to happen?



What's Going On?

- It was intended that the 5% would be added to the amount, returning \$1050
- Was \$1000 the desired output?
- No – so what went wrong?
- This is a very common mistake to make!
 - Let's trace through the code and figure it out

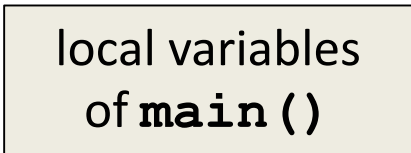
Tracing the Bank Interest Code

- First, we create two variables that are local to `main()`

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

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

local variables
of `main()`

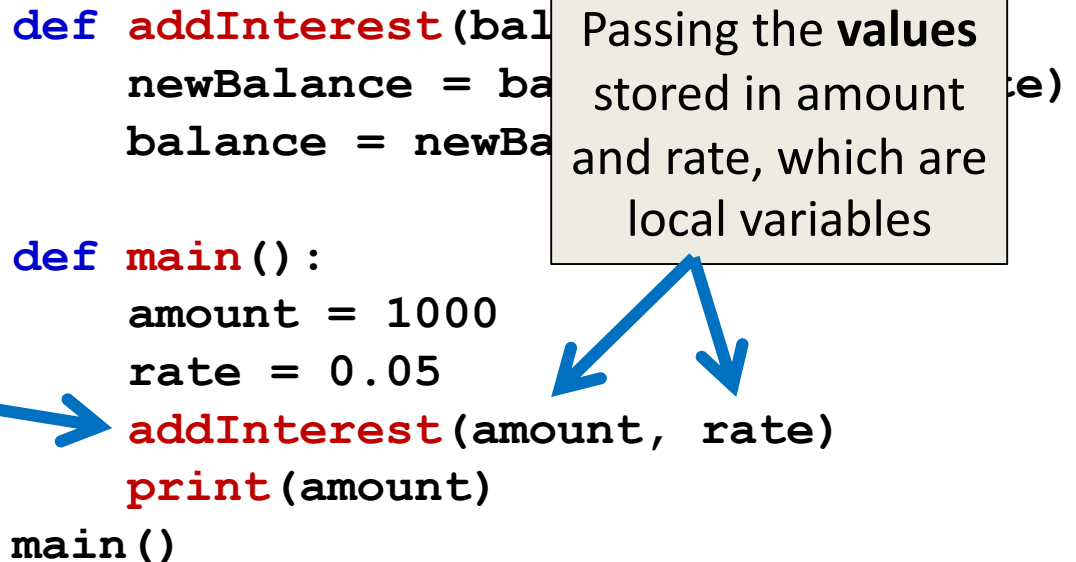


Tracing the Bank Interest Code

- Second, we call `addInterest()` and pass the values of the local variables of `main()` as arguments

```
def addInterest(balance, rate):  
    newBalance = balance + (balance * rate)  
    balance = newBalance  
    return newBalance  
  
def main():  
    amount = 1000  
    rate = 0.05  
    addInterest(amount, rate)  
    print(amount)  
  
main()
```


Passing the values stored in `amount` and `rate`, which are local variables

A diagram illustrating the flow of data in a Python function call. A box on the left labeled "Call to addInterest()" has a blue arrow pointing to the `addInterest(amount, rate)` line in the `main()` function. Another box on the right, labeled "Passing the values stored in amount and rate, which are local variables", has two blue arrows pointing to the `amount` and `rate` arguments in the `addInterest` function call.

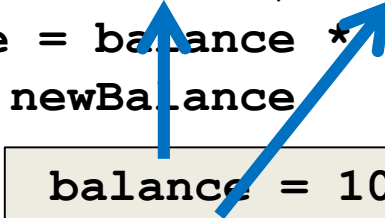
Tracing the Bank Interest Code

- Third, when control is passed to `addInterest()`, the formal parameters (`balance` and `rate`) are set to the value of the arguments (`amount` and `rate`)

Control passes to
`addInterest()`



```
def addInterest(balance, rate):  
    newBalance = balance * (1 + rate)  
    balance = newBalance  
  
def main():  
    amount = 1000  
    rate = 0.05  
    addInterest(amount, rate)  
    print(amount)  
  
main()
```



balance = 1000
rate = 0.05

Tracing the Bank Interest Code

- Even though the parameter **rate** appears in both **main()** and **addInterest()**, they are two separate variables because of scope

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

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

Even though **rate** exists in both **main()** and **addInterest()**, they are in two separate scopes

Scope

- In other words, the *formal parameters* of a function only receive the values of the *arguments*
- The function does not have access to the original variable in `main()`

New Bank Interest Code

```
def addInterest(balance, rate):  
    newBalance = balance * (1 + rate)  
    return newBalance  
  
def main():  
    amount = 1000  
    rate = 0.05  
    amount = addInterest(amount, rate)  
    print(amount)  
main()
```

New Bank Interest Code

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

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

These are the only
parts we changed

Daily emacs Shortcut

- **M + %**
 - (Meta + Shift + 5)
 - Search and replace
 - Keeps correct case! (cat -> dog, Cat -> Dog, CAT -> DOG)
- First, type the thing to search for; hit Enter
- Second, type the thing replace it with; Enter
 - Hit “y” or “n” for each highlighted instance to indicate if you want to replace that one

Announcements

- Project 1 out this evening
 - Read the whole document before you start!
 - Design due by Monday, Oct 22nd at 8:59:59 PM
 - Final due by Monday, Oct 29th at 8:59:59 PM

Image Sources

- Cardboard box:
 - <https://pixabay.com/p-220256/>
- Wooden ship (adapted from):
 - <https://pixabay.com/p-307603/>
- Coconut island (adapted from):
 - <https://pixabay.com/p-1892861/>
- Dollar sign:
 - <https://pixabay.com/p-634901/>