CMSC201
Computer Science I for Majors

Lecture 11 – Functions (Continued)

All materials copyright UMBC unless otherwise noted

Last Class We Covered

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

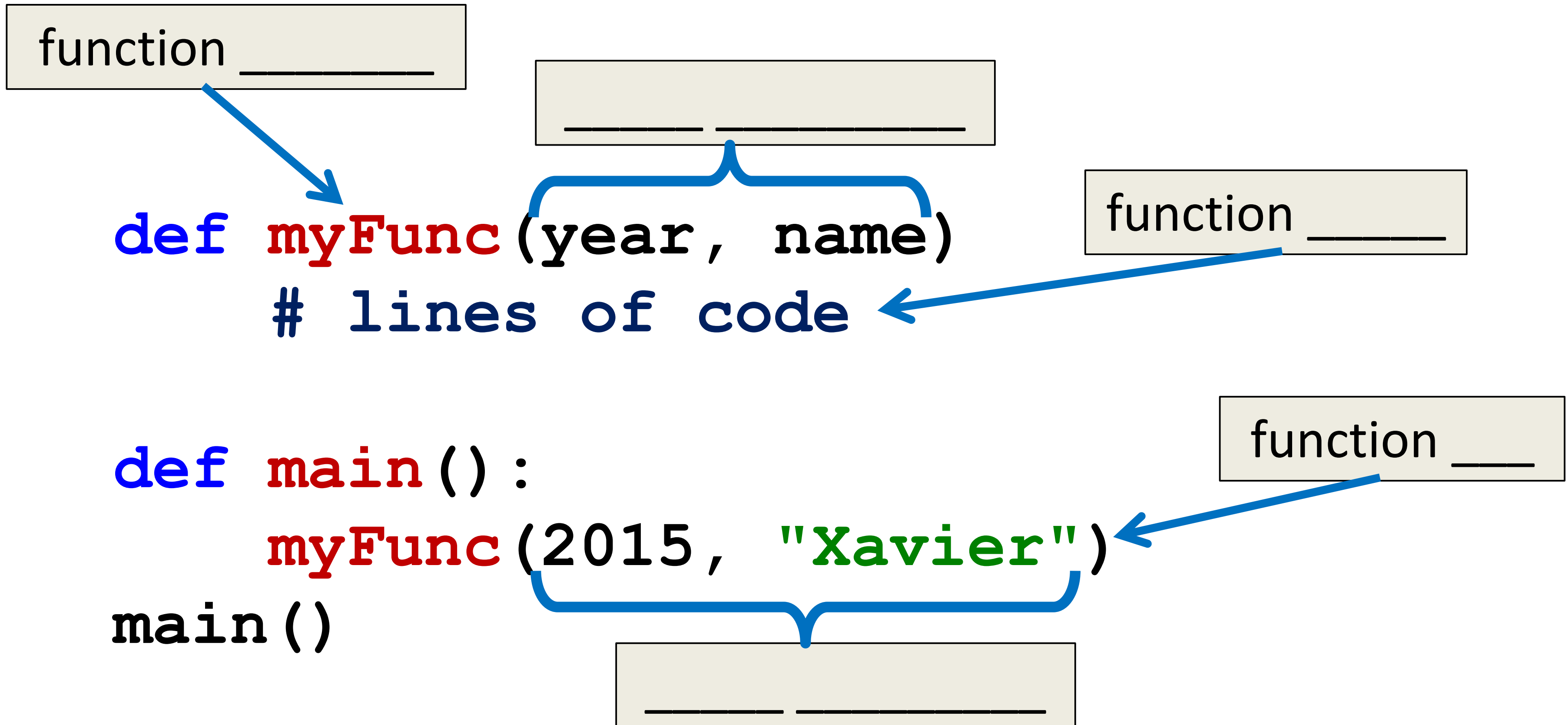
Any Questions from Last Time?

Today's Objectives

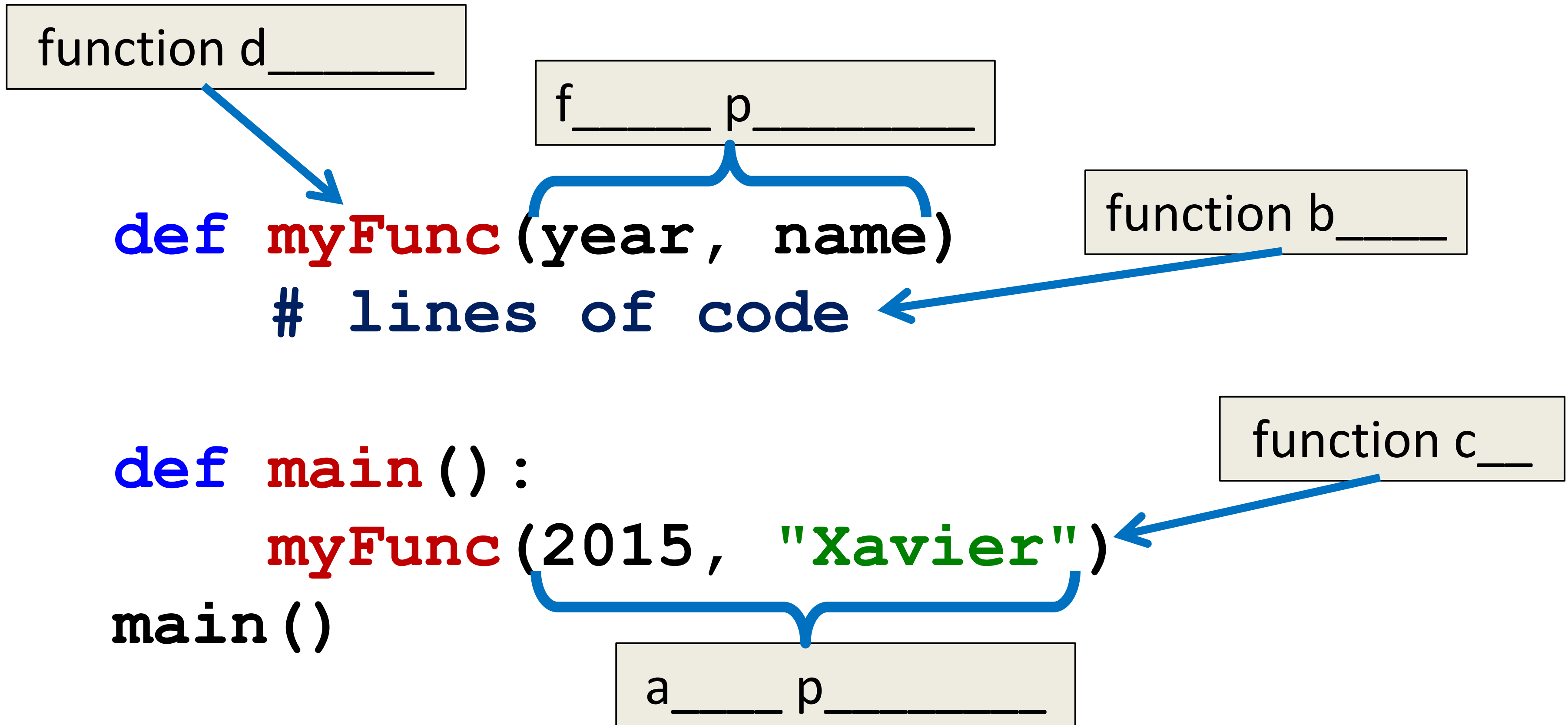
- To introduce value-returning functions
- To understand mutability (and immutability)
  - To better grasp how values in the scope of a function actually work
- To practice function calls and some special situations

Review: Parts of a Function

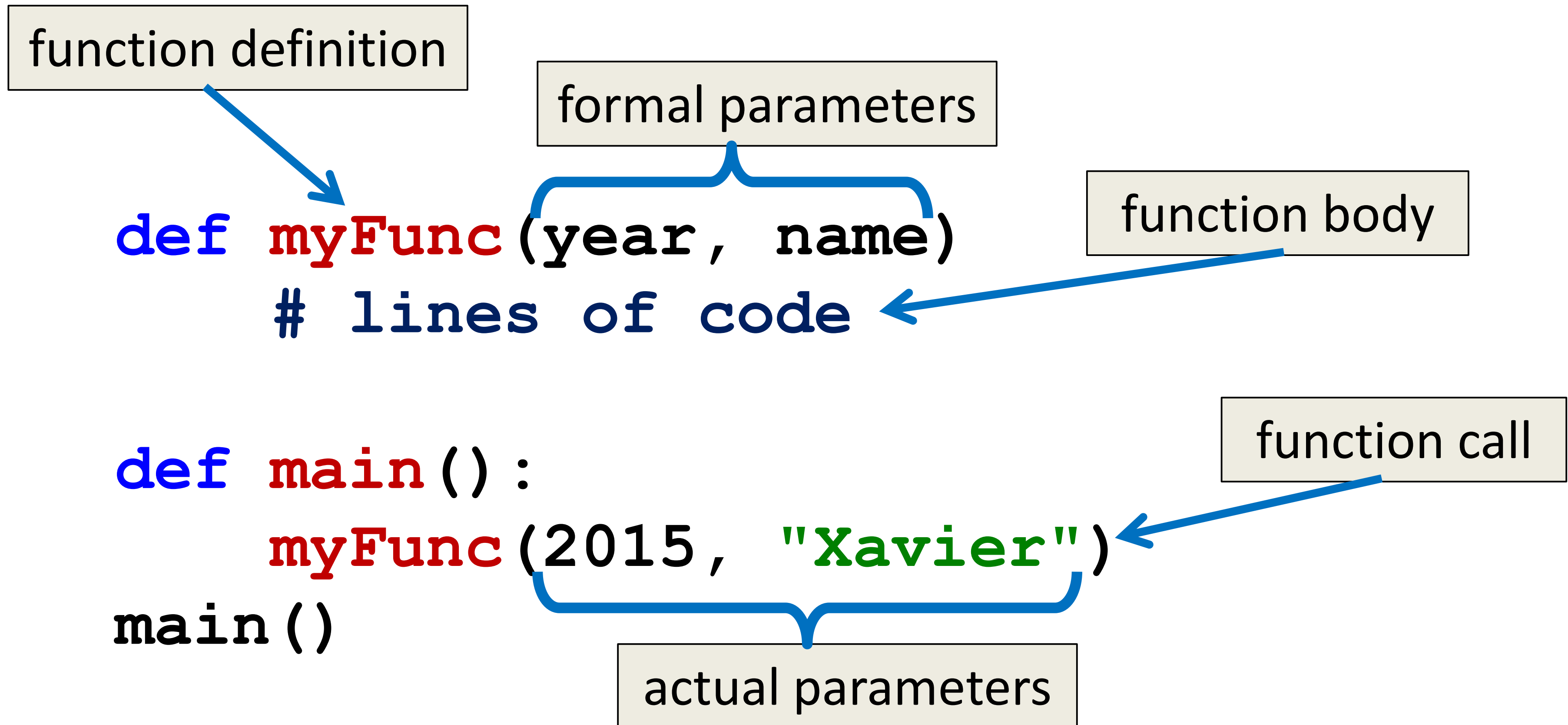
Function Vocabulary



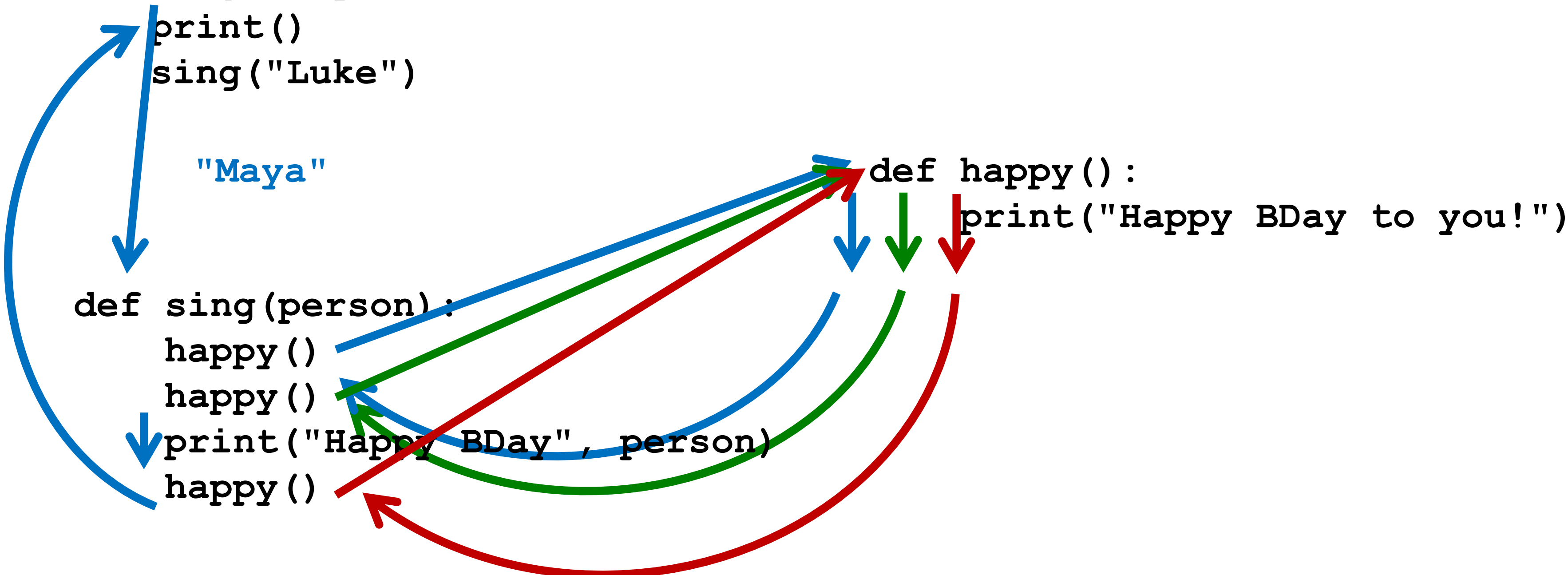
Function Vocabulary



Function Vocabulary



Visual Code Trace



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()` , **`str()`**, **`input()`**, etc.

- For example, **`int()`**
  - Takes in any string as its parameter
  - Processes the digits in the string
  - 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):
    # return the square
    return (num * num)
```

Handling Return Values

- When Python encounters **`return`**, it
  - Exits the function
  - Returns control back to where the function was called
  - Similar to reaching the end of a function

- The value provided in the return statement is sent back to the caller as an ***expression result***

Code Trace: Return from `square()`

Let's follow the flow of the code

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

```
def square(num1):
    return num1 * num1
```

→ Step 1: Call `main()`

Code Trace: Return from `square()`

Let's follow the flow of the code

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

```
def square(num1):
    return num1 * num1
```

Step 1: Call **`main()`**

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

Code Trace: Return from `square()`

Let's follow the flow of the code

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

```
def square(num1):
    return num1 * num1
```

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

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

```
def square(num1):
    return num1 * num1
```

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()
```

```
def square(num1):
    return num1 * num1
```

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()`**

UMBC
AN HONORS UNIVERSITY IN MARYLAND

Code Trace: Return from **square()**

Let's follow the flow of the code

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

```
def square(num1):
    return num1 * num1
```

num1 = 5

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()**
Step 6: Set the value of **num1** in **square()** to **x**

Code Trace: Return from **`square()`**

Let's follow the flow of the code

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

```
def square(num1):
    return num1 * num1
```

`num1 = 5`

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()`**
Step 6: Set the value of **`num1`** in **`square()`** to **`x`**
Step 7: Calculate **`num1 * num1`**

Code Trace: Return from `square()`

Let's follow the flow of the code

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

```
def square(num1):
    return num1 * num1
```

`num1 = 5`

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()**
Step 6: Set the value of **num1** in **square()** to **x**
Step 7: Calculate **num1 * num1**
Step 8: Return to **main()** and set **y** = return statement

Code Trace: Return from square()

Let's follow the flow of the code

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

```
def square(num1):
    return num1 * num1
```

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()**
Step 6: Set the value of **num1** in **square()** to **x**
Step 7: Calculate **num1 * num1**
Step 8: Return to **main()** and set **y** = return statement
Step 9: Print value of **y**

Testing: Return from `square()`

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

Functions with Multiple Return Values

Returning Multiple Values

- Sometimes a function needs to return more than one value
- To do this, simply list more than one expression in the the **`return`** statement

```
def sumDiff(num1, num2):
    sum  = num1 + num2
    diff = num1 - num2
    return sum, diff
```

Accepting Multiple Values

- When calling a function with multiple returns, the code must also use multiple assignments
- Assignment is based on position, just like passing in parameters is based on position

```
sum, diff = sumDiff(xVal, yVal)
```

Accepting Multiple Values

```
def main():
    first  = int(input("Enter first number:  "))
    second = int(input("Enter second number: "))
    sum, diff = sumDiff(first, second)
    print("The sum is", sum,
          "and the difference is", diff)

def sumDiff(num1, num2):
    theSum = num1 + num2
    theDiff = num1 - num2
    return theSum, theDiff

main()
```

Accepting Multiple Values

sum gets the first value returned

diff gets the second value returned

```
        = int(inp                        ber:  "))
    second = int(input("Enter second number: "))
    sum, diff = sumDiff(first, second)
    print("The sum is", sum,
          "and the difference is", diff)

def sumDiff(num1, num2):
    theSum = num1 + num2
    theDiff = num1 - num2
    return theSum, theDiff

main()
```

Accepting Multiple Values

```
def main():
    first  = int(input("Enter first number:  "))
    second = int(input("Enter second number: "))
    sum, diff = sumDiff(first, second)
    print("The sum is", sum,
          "and the differenc

def sumDiff(num1, num2):
    theSum = num1 + num2
    theDiff = num1 - num2
    return theSum, theDiff

main()
```

Notice that none of the variable names match!

Variable names do not need to match when calling a function.

Remember scope!

Every Function Returns *Something*

- All Python functions return a value
  - Even if they don’t have a **`return`** statement

- Functions without an explicit **`return`** hand back a special object, called **`None`**
  - **`None`** is the absence of a value

Common Errors and Problems

- Writing a function that returns a value but…
- Forgetting to include the **`return`** statement

```
>>> def test():
...     print("In the fxn")
...     var = 3
>>> var2 = test()
In the fxn
>>> print(var2)
None
```

Variable assigned to the return value will be **`None`**.

Common Errors and Problems

- Writing a function that returns a value but…
- Forgetting to assign that value to anything

```
>>> def test():
...     print("In the fxn")
...     return 3
>>> var2 = 7
>>> test()
In the fxn
>>> print(var2)
7
```

The variable **var2** was not updated; the code should have read **var2 = test()**

Common Errors and Problems

- Writing a function that returns value(s) but…

- Not assigning the right number of variables

```
>>> def test():
...       print("In the fxn")
...       return 3
>>> var1, var2 = test()
In the fxn
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not iterable
```

Common Errors and Problems

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

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

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

Modifying Parameters

Other Ways to Pass Back Information

- A `return` value is the main way to send information back from a function
- We may also be able to pass information back by making changes directly to the parameters
- One of the problems with modifying parameters is due to ***scope***

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

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

What is the output of this code?

```
1000
```

Is this what we expected?

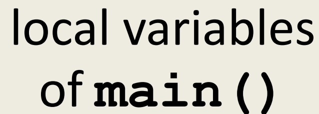
 Image from pixabay.com

What's Going On?

- We thought that the 5% would be added to the amount, returning $1050
- Was $1000 the expected output?
- No – so what went wrong?
  - Let's trace through the program and find out

Tracing the Bank Interest Code

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

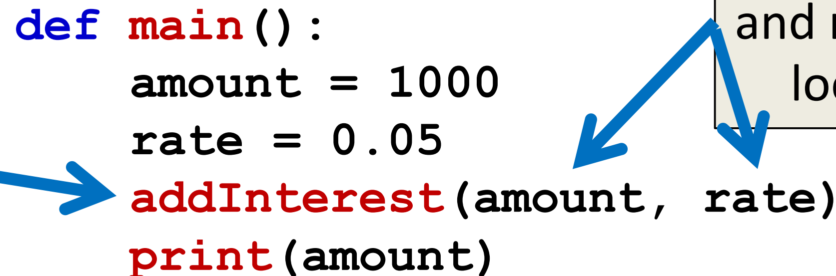



```
def main():
    amount = 1000
    rate = 0.05
    addInterest(amount, rate)
    print(amount)

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

Tracing the Bank Interest Code

- Second, we call **`addInterest()`** and pass the local variables of **`main()`** as actual parameters




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

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

Tracing the Bank Interest Code

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

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

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

Control passes to **addInterest()**

```
balance = amount
   rate = rate
```

Tracing the Bank Interest Code

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

Even though rate is in both **`main()`** and **`addInterest()`**, they are in different places in memory

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

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

Scope

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

Mutability

Mutable and Immutable

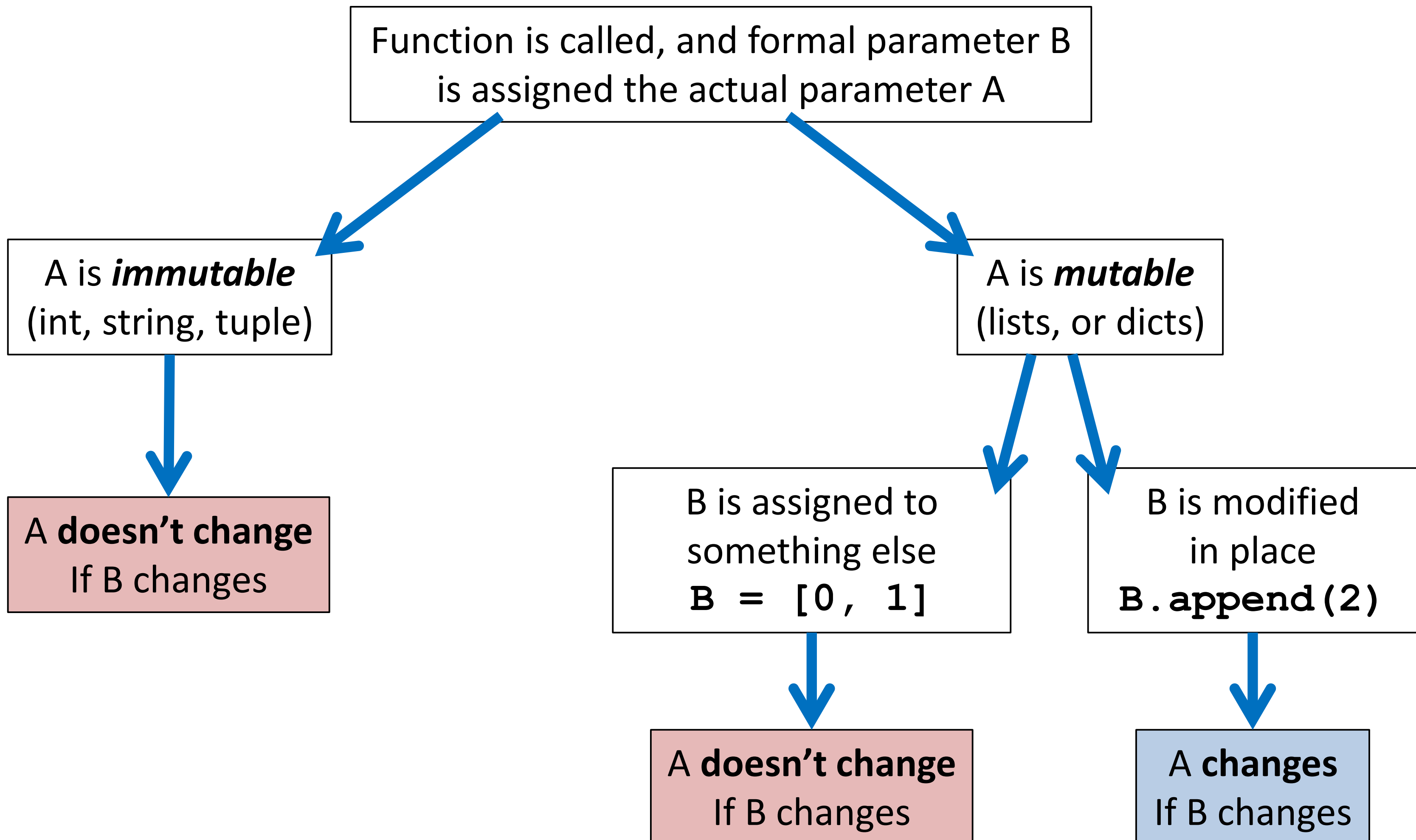
- In python, certain structures cannot be altered once they are created and are called ***immutable***
  - These include integers, strings, and tuples

- Other structures can be altered after they are created and are called ***mutable***
  - These include lists and dictionaries

Scope and Mutability in Functions

- To get a better idea for how this works with functions, let's look at an example
- We can call a function with actual parameters that are mutable or that are immutable
- When we alter the formal parameters in the function, we could overwrite, or we could update it (change the parameter in place)

Scope and Mutability in Functions



Scope and Mutability in Functions

- A good general rule for if it will be altered:

- When you use the ***assignment operator***, the parameter won't actually be changed in **`main()`**
  - Unless you are editing one element, like in a list

- When you use something like **`.append()`** on the parameter, it will be changed in **`main()`**

The Bank Interest Example

Updating Bank Interest

- The variable we wanted to update, **`balance`**, is a float, which means it is...
  - Immutable

- We can't change it from within the function
- What other options do we have?
  - Change the function so it returns a **`newBalance`**

New Bank Interest Code

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

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

New Bank Interest Code

These are the only parts we changed

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

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

New Bank Interest Code Trace

Let's follow the flow of the code

```
def addInt(balance, rate):
    newBal = balance * (1 + rate)
    return newBal
```

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

Once we leave **addInt()**, the values of balance and rate are removed from memory

Step 1: Call **main()**
Step 2: Pass control to **def main()**
Step 3: Set **amount = 1000** and **rate = 0.05**
Step 4: Set amount = return statement of **addInt()**
Step 5: Pass control from **main()** to **addInt()**
Step 6: Set the value of **balance** in **addInt()** to amount
Step 7: Set the value of rate in **addInt()** to rate
Step 8: Set value of **newBal** to **balance * (1 + rate)**
Step 9: Return to **main()** and set value of **amount = newBal**
Step 10: Print value of **amount**

Passing Lists to Functions

Multiple Bank Accounts

- Instead of a single account, we are writing a program for a bank that has many accounts
  - We could store the account balances in a list, then update the interest for each balance in the list

- We could update the first balance in the list with code like:

```
balances[0] = balances[0] * (1 + rate)
```

Multiple Bank Accounts

```
balances[0] = balances[0] * (1 + rate)
```

- This code says, “multiply the value in the 0th position of the list by (1 + rate) and store the result back into the 0th position of the list”
- A more general way to do this would be with a loop that goes through the indexes from **`0`**, **`1`**, …, **`length - 1`**

Example: Multiple Interest

```
# addinterest3.py
# Illustrates a mutable parameter (a list)

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

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

main()
```

Multiple Interest Output

- Our original code had these values:

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

- The program returns:

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

- Because **`balances`** is a list, and we are updating it in place, so the actual values are changed

Announcements

- Homework 5 is due Wednesday
  - Homework 3 grades went out Sunday night
- Homework 6 does <u>not</u> come out this week
  - It will come out the night of October 20th
- The midterm exam is when?
  - During class on October 19th and 20th!