Objectives

- Defining your own functions
  - Control flow
  - Scope, variable lifetime
- Programmatically testing functions

Review

- How do we create a file object?
- What are the two ways to read from a file?
- What should we always do after we create/open a file object?

Looking behind the curtain…

DEFINING OUR OWN FUNCTIONS

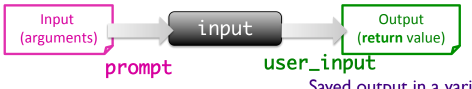
Functions

- We've used functions
  - Built-in functions: `len`, `input`, `eval`
  - Functions from modules, e.g., `math` and `random`
- Benefits
  - Reuse, reduce code
  - Easier to read, write (because of ***abstraction***)

Today, we'll learn how to
define our own functions!

Review: Functions

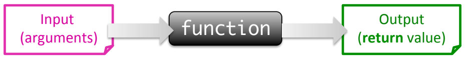
- Function is a **black box**
  - Implementation doesn't matter
  - Only care that function generates appropriate output, given appropriate input
- Example:
  - Didn't care how `input` function was implemented
  - Use: `user_input = input(prompt)`



Saved output in a variable

Creating Functions

- A function can have
  - 0 or more inputs
  - 0 or 1 outputs
- When we define a function, we know its inputs and if it has output



Why Write Functions?

- Allows you to break up a hard problem into *smaller*, more *manageable* parts
- Makes your code easier to *understand*
- Hides implementation details (*abstraction*)
  - Provides interface (input, output)
- Makes part of the code *reusable* so that you:
  - Only have to write function code once
  - Can debug it all at once
    - Isolates errors
  - Can make changes in one function (*maintainability*)

Similar to benefits of OO Programming

Writing a Function

- I want a function that averages two numbers

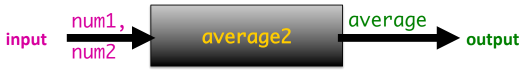
• What is the input to this function?
• What is the output to this function?

Writing a Function

- I want a function that averages two numbers
- What is the input to this function?
  - The two numbers
- What is the output to this function?
  - The average of those two numbers, as a float

These are key questions to ask yourself when designing your own functions.
• Inputs: parameters?
• Output: what is getting returned?

Averaging Two Numbers



- Input: the two numbers
- Output: the average of two numbers

Syntax of Function Definition

Keyword — Function Name — Input Name/ Parameter

```
def average2(num1, num2):
    """
    Parameters: two numbers to be averaged.
    Returns the average of two numbers
    """

    average = (num1 + num2)/2
    return average
```

Function header

Function documentation

Body (or function definition)

Keyword: How to give output — Output

Calling your own functions

Same as calling someone else's functions …

```
average = average2(100, 4.34)
```

Output is assigned to average — Function Name — Input

average2.py

Functions: Similarity to Math

- In math, a function definition looks like:

  $f(x) = x^2 + 2$

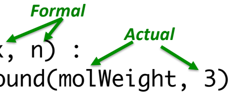
- Plug values in for x
- Example:
  - $f(3) = 3^2 + 2 = 11$
  - 3 is your ***input***, assigned to x
  - 11 is output

Parameters

- The inputs to a function are called ***parameters*** or ***arguments***, depending on the context
- When ***calling***/using functions, arguments must appear in same order as in the function header
  - Example: `round(x, n)`
    - **x** is the `float` to round
    - **n** is `int` of decimal places to round **x** to

Parameters

- **Formal Parameters** are the variables named in the function definition
- **Actual Parameters** or **Arguments** are the variables or literals that really get used when the function is called.



Defined: `def round(x, n) :`

Use: `roundCelc = round(molWeight, 3)`

Formal & actual parameters must match in ***order***, ***number***, and ***type***!

Passing Parameters

- Only **copies** of the actual parameters are given to the function for **immutable** data types
  - Immutable types: most of what we've talked about so far
    - Strings, integers, floats
  - The actual parameters in the *calling* code **do *not*** change
- (Lists are mutable and have different rules)

Function Output

- When the code reaches a statement like

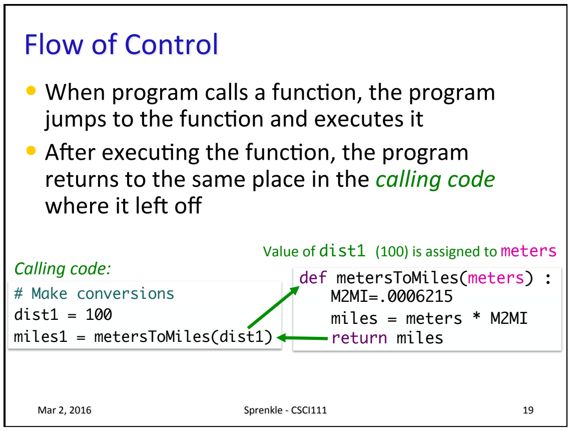
  ```
  return x
  ```

  - The function stops executing
  - `x` is the **output *returned*** to the place where the function was called
- For functions that don't have explicit output, `return` does not have a value with it, e.g.,

  ```
  return
  ```

- Optional: don't *need* to have `return`
  - Function *automatically* returns at the end

CONTROL FLOW WITH FUNCTIONS



Flow of Control

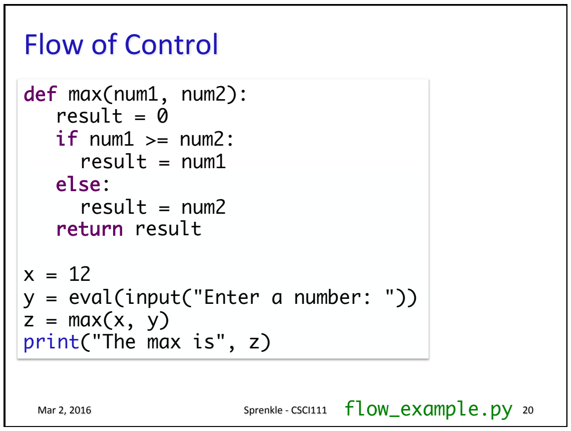
- When program calls a function, the program jumps to the function and executes it
- After executing the function, the program returns to the same place in the *calling code* where it left off

Value of `dist1` (100) is assigned to `meters`

Calling code:

```
# Make conversions
dist1 = 100
miles1 = metersToMiles(dist1)
```

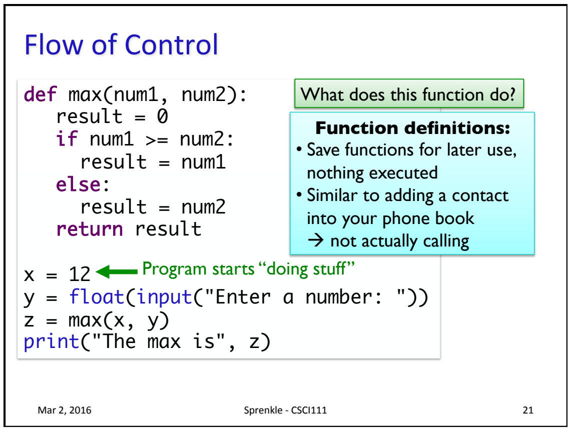
```
def metersToMiles(meters) :
   M2MI=.0006215
   miles = meters * M2MI
   return miles
```



Flow of Control

```
def max(num1, num2):
   result = 0
   if num1 >= num2:
      result = num1
   else:
      result = num2
   return result

x = 12
y = eval(input("Enter a number: "))
z = max(x, y)
print("The max is", z)
```

flow_example.py



Flow of Control

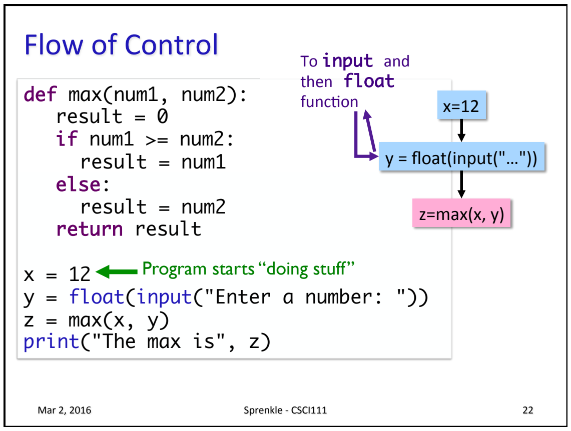
What does this function do?

```
def max(num1, num2):
   result = 0
   if num1 >= num2:
      result = num1
   else:
      result = num2
   return result

x = 12
y = float(input("Enter a number: "))
z = max(x, y)
print("The max is", z)
```

Function definitions:

- Save functions for later use, nothing executed
- Similar to adding a contact into your phone book → not actually calling

Program starts "doing stuff"



Flow of Control

```
def max(num1, num2):
   result = 0
   if num1 >= num2:
      result = num1
   else:
      result = num2
   return result

x = 12
y = float(input("Enter a number: "))
z = max(x, y)
print("The max is", z)
```

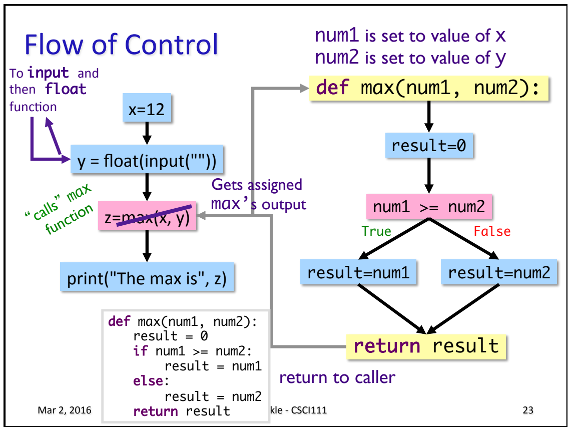
Program starts "doing stuff"

To `input` and then `float` function

x=12

y = float(input("…"))

z=max(x, y)



Flow of Control

num1 is set to value of x
num2 is set to value of y

To `input` and then `float` function

x=12

y = float(input(""))

"calls" max function

z=~~max~~(x, y)

Gets assigned max's output

print("The max is", z)

`def max(num1, num2):`

result=0

num1 >= num2

True

False

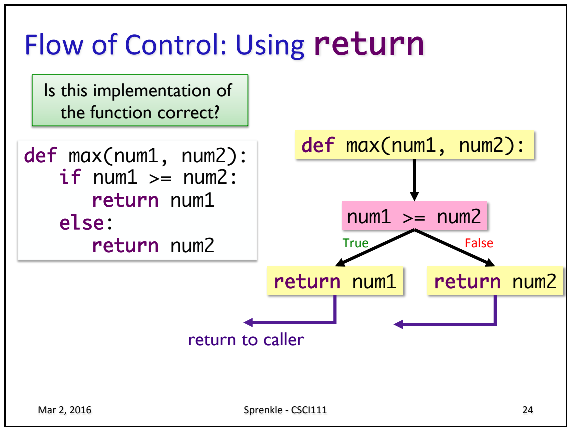
result=num1

result=num2

`return result`

return to caller

```
def max(num1, num2):
   result = 0
   if num1 >= num2:
      result = num1
   else:
      result = num2
   return result
```



Flow of Control: Using `return`

Is this implementation of the function correct?

```
def max(num1, num2):
   if num1 >= num2:
      return num1
   else:
      return num2
```

`def max(num1, num2):`

num1 >= num2

True

False

`return num1`

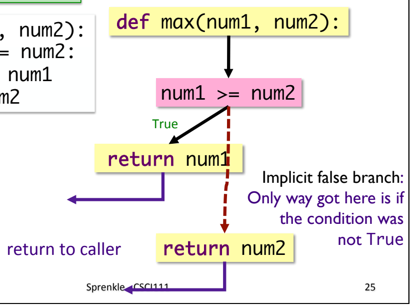
`return num2`

return to caller

Flow of Control: Using return

Is this implementation of the function correct?

```
def max(num1, num2):
    if num1 >= num2:
        return num1
    return num2
```



Implicit false branch: Only way got here is if the condition was not True

Function Input and Output

- What does this function do?
- Identify function's input and output

```
def printVerse(animal, sound):
    print(BEGIN_END + EIEIO)
    print("And on that farm he had a " + animal + EIEIO)
    print("With a " + sound + ", " + sound + " here")
    print("And a " + sound + ", " + sound + " there")
    print("Here a", sound)
    print("There a", sound)
    print("Everywhere a " + sound + ", " + sound)
    print(BEGIN_END + EIEIO)
    print()
```

Function Input and Output

- 2 inputs: animal and sound
- 0 outputs
  - ***Displays*** something but does not return anything (None)

```
def printVerse(animal, sound):
    print(BEGIN_END + EIEIO)
    print("And on that farm he had a " + animal + EIEIO)
    print("With a " + sound + ", " + sound + " here")
    print("And a " + sound + ", " + sound + " there")
    print("Here a", sound)
    print("There a", sound)
    print("Everywhere a " + sound + ", " + sound)
    print(BEGIN_END + EIEIO)
    print()
```

Function exits here

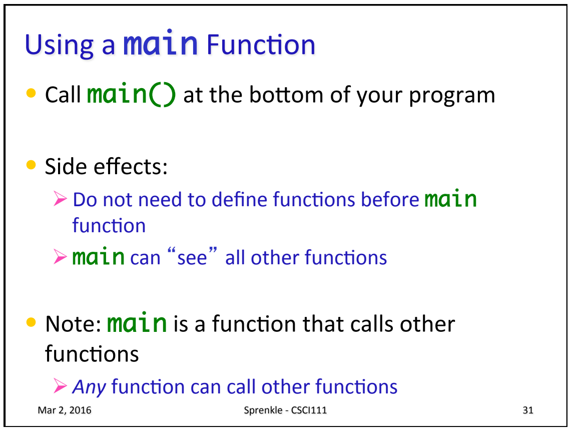
PROGRAM ORGANIZATION

Where are Functions Defined?

- Functions can go inside program script
  - If no main() function, defined ***before*** use/called
    - average2.py
  - If main() function, defined anywhere in script
- Functions can go inside a separate ***module***

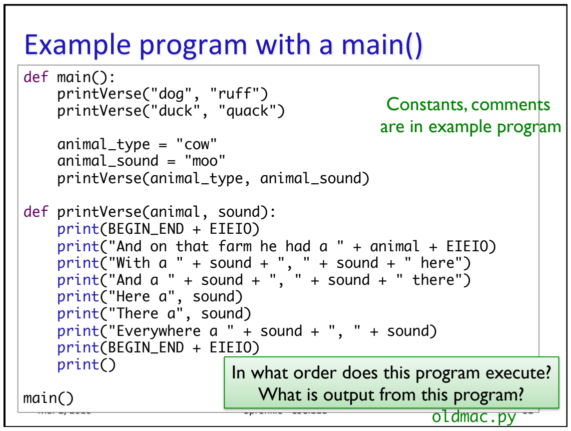
Program Organization: main function

- In many languages, you put the “driver” for your program in a main function
  - You can (and should) do this in Python as well
- Typically main functions are defined at the top of your program
  - Readers can quickly see an overview of what program does
- main usually takes no arguments
  - Example: `def main():`



Using a main Function

- Call main() at the bottom of your program
- Side effects:
  - Do not need to define functions before main function
  - main can "see" all other functions
- Note: main is a function that calls other functions
  - *Any* function can call other functions



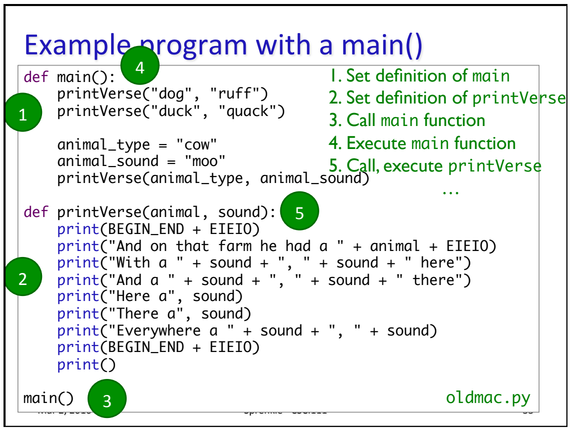
Example program with a main()

```
def main():
    printVerse("dog", "ruff")
    printVerse("duck", "quack")

    animal_type = "cow"
    animal_sound = "moo"
    printVerse(animal_type, animal_sound)

def printVerse(animal, sound):
    print(BEGIN_END + EIEIO)
    print("And on that farm he had a " + animal + EIEIO)
    print("With a " + sound + ", " + sound + " here")
    print("And a " + sound + ", " + sound + " there")
    print("Here a", sound)
    print("There a", sound)
    print("Everywhere a " + sound + ", " + sound)
    print(BEGIN_END + EIEIO)
    print()

main()
```

Constants, comments are in example program

In what order does this program execute? What is output from this program?

oldmac.py



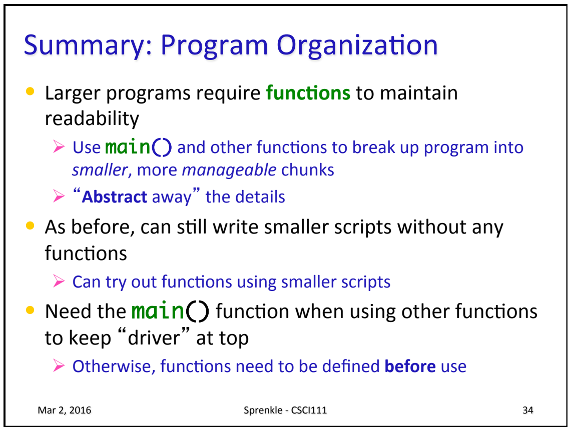
Example program with a main()

```
def main():
    printVerse("dog", "ruff")
    printVerse("duck", "quack")

    animal_type = "cow"
    animal_sound = "moo"
    printVerse(animal_type, animal_sound)

def printVerse(animal, sound):
    print(BEGIN_END + EIEIO)
    print("And on that farm he had a " + animal + EIEIO)
    print("With a " + sound + ", " + sound + " here")
    print("And a " + sound + ", " + sound + " there")
    print("Here a", sound)
    print("There a", sound)
    print("Everywhere a " + sound + ", " + sound)
    print(BEGIN_END + EIEIO)
    print()

main()
```

1. Set definition of main
2. Set definition of printVerse
3. Call main function
4. Execute main function
5. Call, execute printVerse

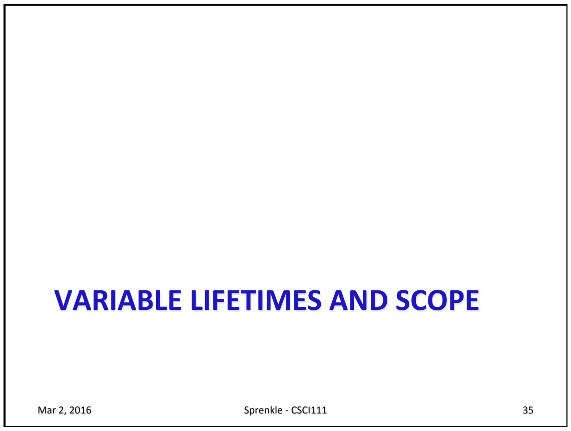
…

oldmac.py

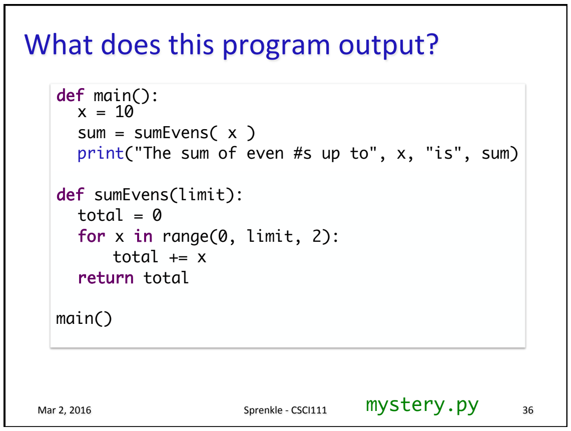


Summary: Program Organization

- Larger programs require **functions** to maintain readability
  - Use main() and other functions to break up program into *smaller*, more *manageable* chunks
  - "**Abstract** away" the details
- As before, can still write smaller scripts without any functions
  - Can try out functions using smaller scripts
- Need the main() function when using other functions to keep "driver" at top
  - Otherwise, functions need to be defined **before** use



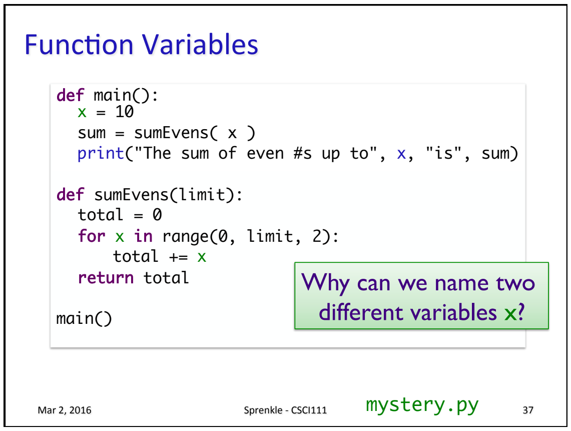
VARIABLE LIFETIMES AND SCOPE



What does this program output?

```
def main():
   x = 10
   sum = sumEvens( x )
   print("The sum of even #s up to", x, "is", sum)

def sumEvens(limit):
   total = 0
   for x in range(0, limit, 2):
       total += x
   return total

main()
```

mystery.py

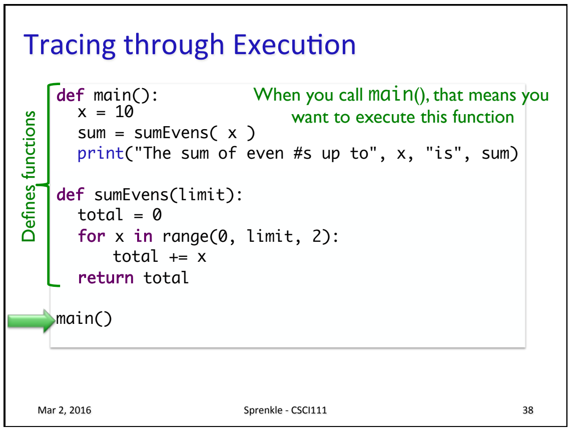


Function Variables

```
def main():
  x = 10
  sum = sumEvens( x )
  print("The sum of even #s up to", x, "is", sum)

def sumEvens(limit):
  total = 0
  for x in range(0, limit, 2):
     total += x
  return total

main()
```

Why can we name two different variables x?

mystery.py

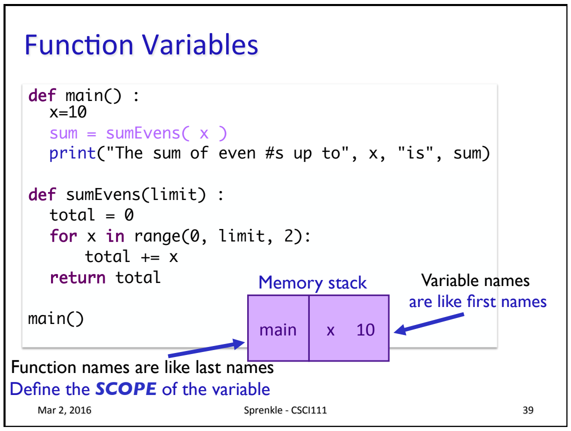


Tracing through Execution

```
def main():
  x = 10
  sum = sumEvens( x )
  print("The sum of even #s up to", x, "is", sum)

def sumEvens(limit):
  total = 0
  for x in range(0, limit, 2):
     total += x
  return total

main()
```

Defines functions

When you call main(), that means you want to execute this function



Function Variables

```
def main() :
  x=10
  sum = sumEvens( x )
  print("The sum of even #s up to", x, "is", sum)

def sumEvens(limit) :
  total = 0
  for x in range(0, limit, 2):
     total += x
  return total

main()
```

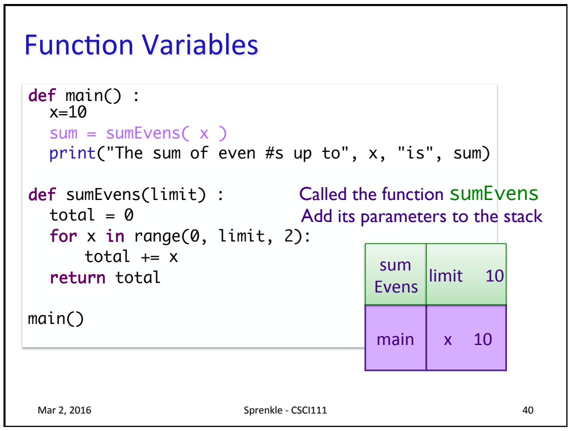
Memory stack

main	x 10

Variable names are like first names

Function names are like last names

Define the ***SCOPE*** of the variable



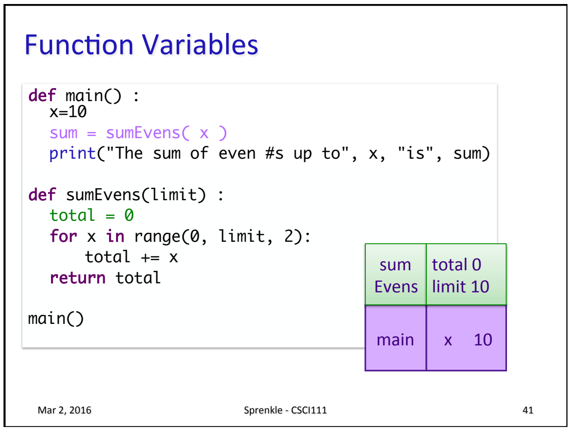
Function Variables

```
def main() :
  x=10
  sum = sumEvens( x )
  print("The sum of even #s up to", x, "is", sum)

def sumEvens(limit) :
  total = 0
  for x in range(0, limit, 2):
     total += x
  return total

main()
```

Called the function sumEvens

Add its parameters to the stack

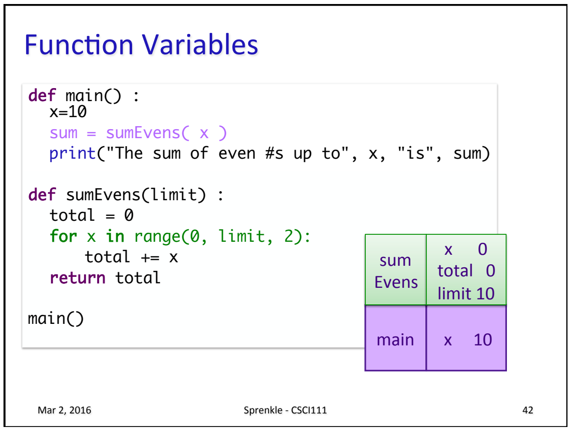
sum Evens	limit 10
main	x 10



Function Variables

```
def main() :
  x=10
  sum = sumEvens( x )
  print("The sum of even #s up to", x, "is", sum)

def sumEvens(limit) :
  total = 0
  for x in range(0, limit, 2):
     total += x
  return total

main()
```

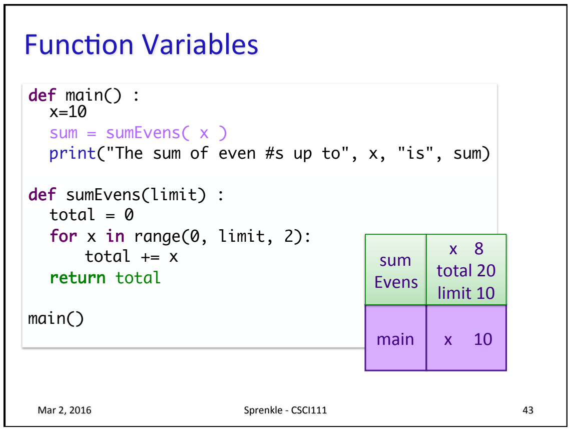
sum Evens	total 0 limit 10
main	x 10



Function Variables

```
def main() :
  x=10
  sum = sumEvens( x )
  print("The sum of even #s up to", x, "is", sum)

def sumEvens(limit) :
  total = 0
  for x in range(0, limit, 2):
     total += x
  return total

main()
```

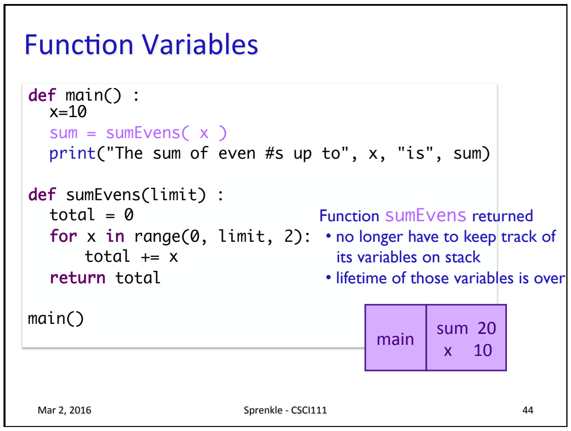
sum Evens	x 0 total 0 limit 10
main	x 10



Function Variables

```
def main() :
   x=10
   sum = sumEvens( x )
   print("The sum of even #s up to", x, "is", sum)

def sumEvens(limit) :
   total = 0
   for x in range(0, limit, 2):
        total += x
   return total

main()
```

sum Evens	x 8 total 20 limit 10
main	x 10



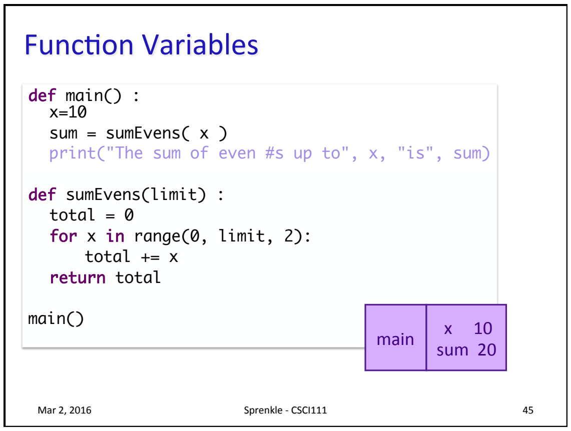
Function Variables

```
def main() :
   x=10
   sum = sumEvens( x )
   print("The sum of even #s up to", x, "is", sum)

def sumEvens(limit) :
   total = 0
   for x in range(0, limit, 2):
        total += x
   return total

main()
```

Function sumEvens returned

- no longer have to keep track of its variables on stack
- lifetime of those variables is over

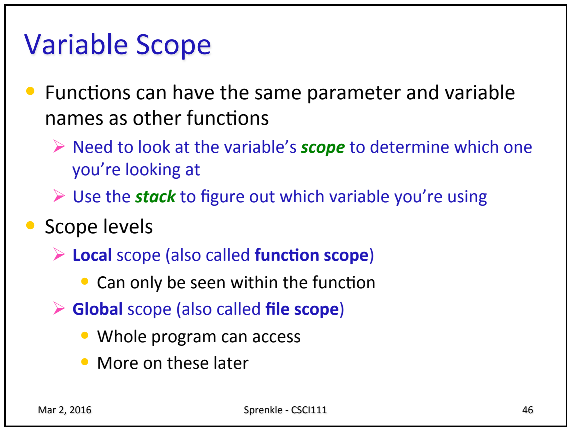
main	sum 20 x 10



Function Variables

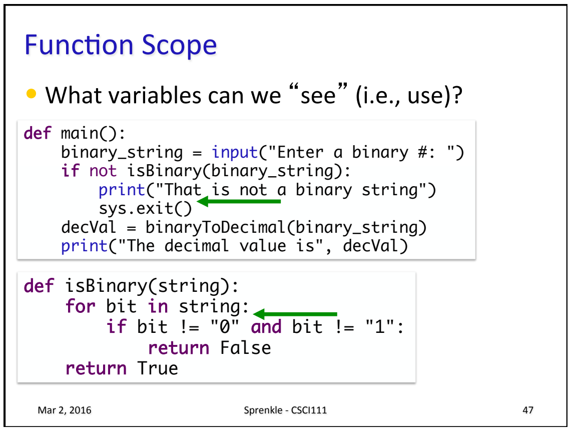
```
def main() :
   x=10
   sum = sumEvens( x )
   print("The sum of even #s up to", x, "is", sum)

def sumEvens(limit) :
   total = 0
   for x in range(0, limit, 2):
        total += x
   return total

main()
```

main	x 10 sum 20



Variable Scope

- Functions can have the same parameter and variable names as other functions
  - Need to look at the variable's *scope* to determine which one you're looking at
  - Use the ***stack*** to figure out which variable you're using
- Scope levels
  - **Local** scope (also called **function scope**)
    - Can only be seen within the function
  - **Global** scope (also called **file scope**)
    - Whole program can access
    - More on these later

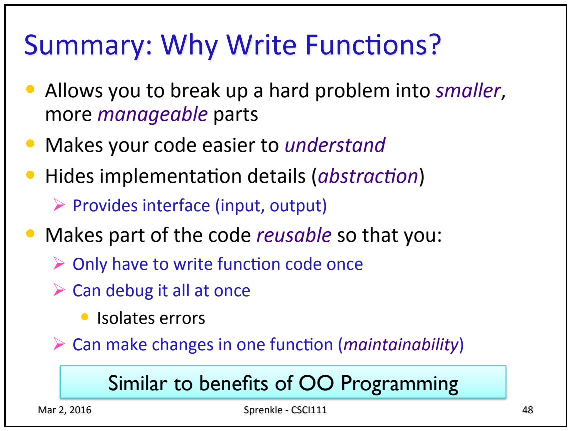


Function Scope

- What variables can we "see" (i.e., use)?

```
def main():
    binary_string = input("Enter a binary #: ")
    if not isBinary(binary_string):
        print("That is not a binary string")
        sys.exit()
    decVal = binaryToDecimal(binary_string)
    print("The decimal value is", decVal)
```

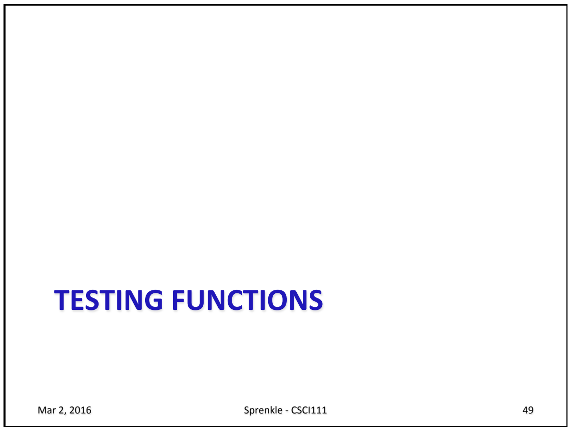
```
def isBinary(string):
    for bit in string:
        if bit != "0" and bit != "1":
            return False
    return True
```



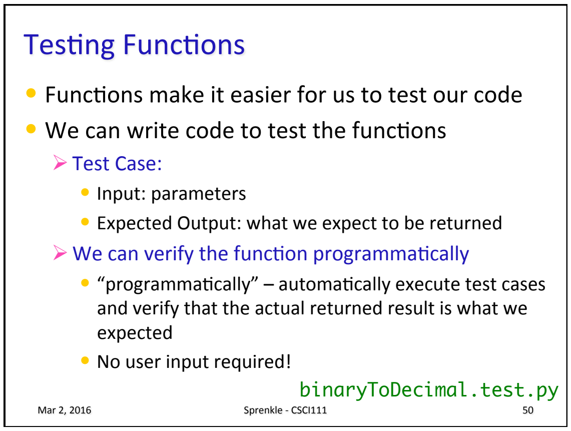
Summary: Why Write Functions?

- Allows you to break up a hard problem into *smaller*, more *manageable* parts
- Makes your code easier to *understand*
- Hides implementation details (*abstraction*)
  - Provides interface (input, output)
- Makes part of the code *reusable* so that you:
  - Only have to write function code once
  - Can debug it all at once
    - Isolates errors
  - Can make changes in one function (*maintainability*)

Similar to benefits of OO Programming



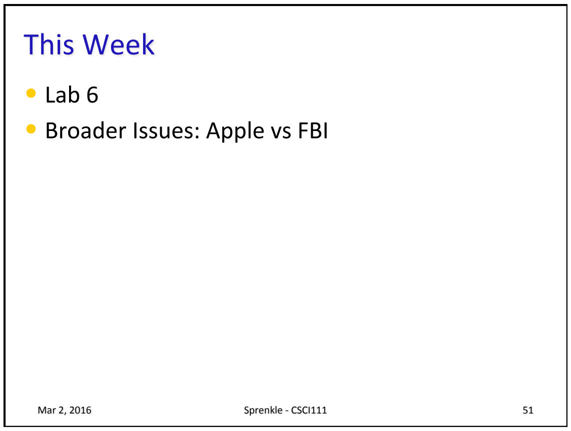
TESTING FUNCTIONS



Testing Functions

- Functions make it easier for us to test our code
- We can write code to test the functions
  - Test Case:
    - Input: parameters
    - Expected Output: what we expect to be returned
  - We can verify the function programmatically
    - “programmatically” – automatically execute test cases and verify that the actual returned result is what we expected
    - No user input required!

`binaryToDecimal.test.py`



This Week

- Lab 6
- Broader Issues: Apple vs FBI