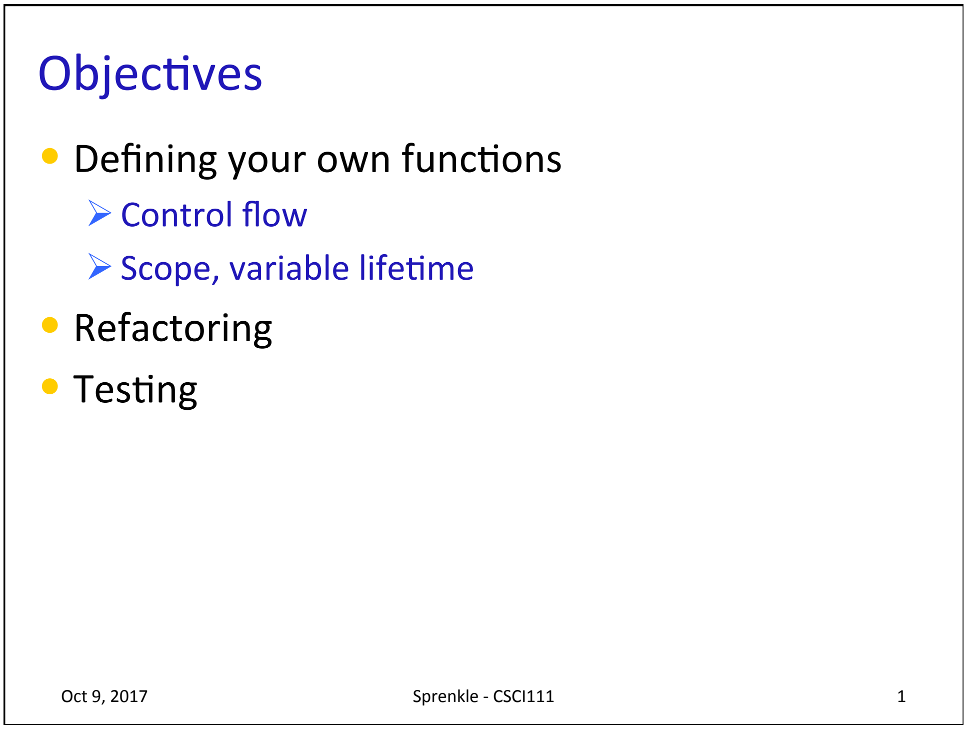
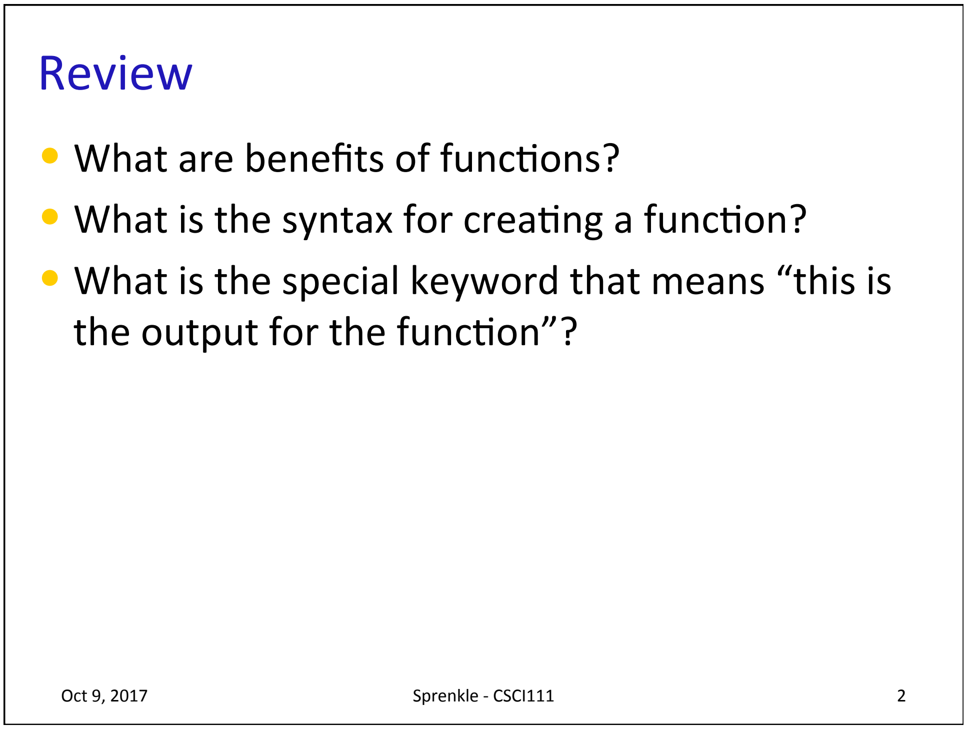


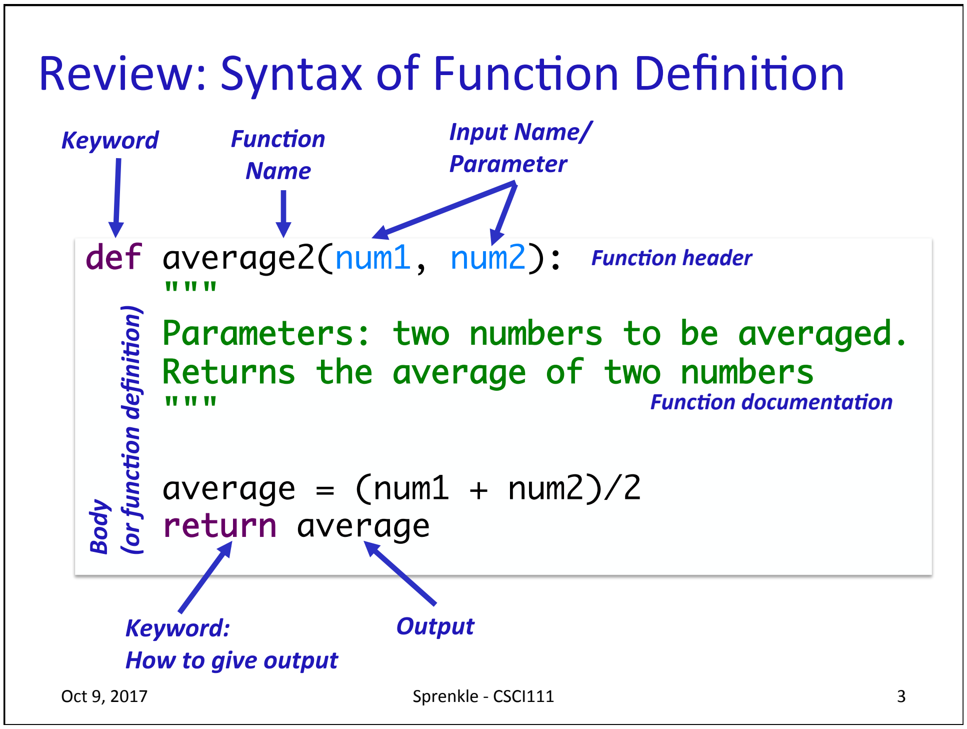
Objectives

- Defining your own functions
  - Control flow
  - Scope, variable lifetime
- Refactoring
- Testing



Review

- What are benefits of functions?
- What is the syntax for creating a function?
- What is the special keyword that means “this is the output for the function”?



Review: 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
```

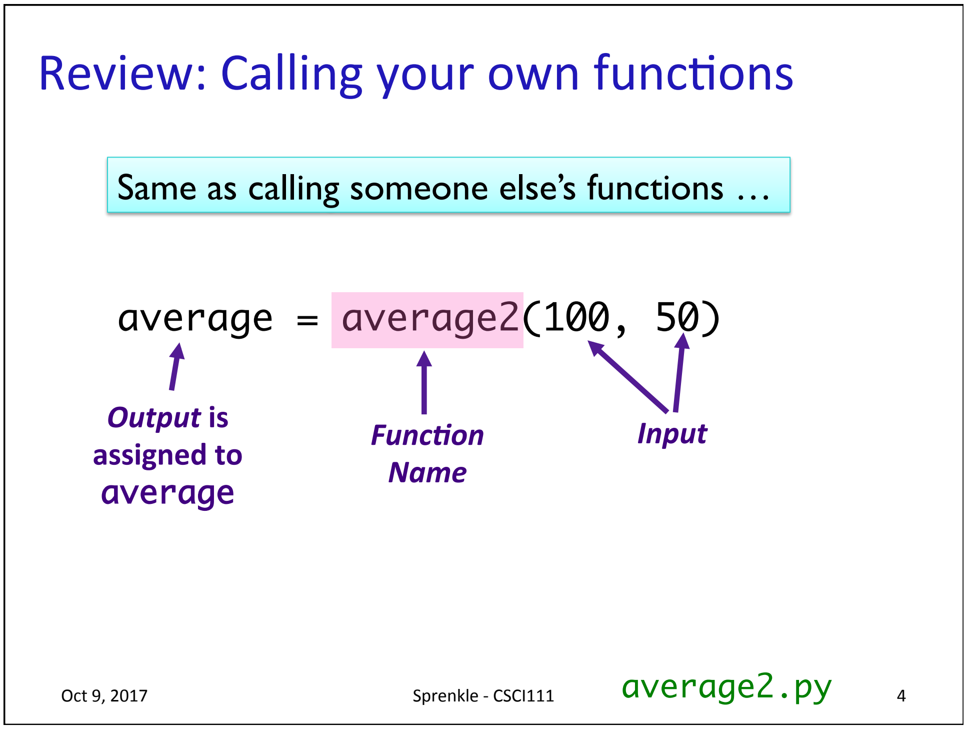
Function header

Function documentation

Body (or function definition)

Keyword: How to give output

Output



Review: Calling your own functions

Same as calling someone else's functions …

```
average = average2(100, 50)
```

***Output* is assigned to average**

Function Name

Input

average2.py

Review: 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

Review: Example program with a main()

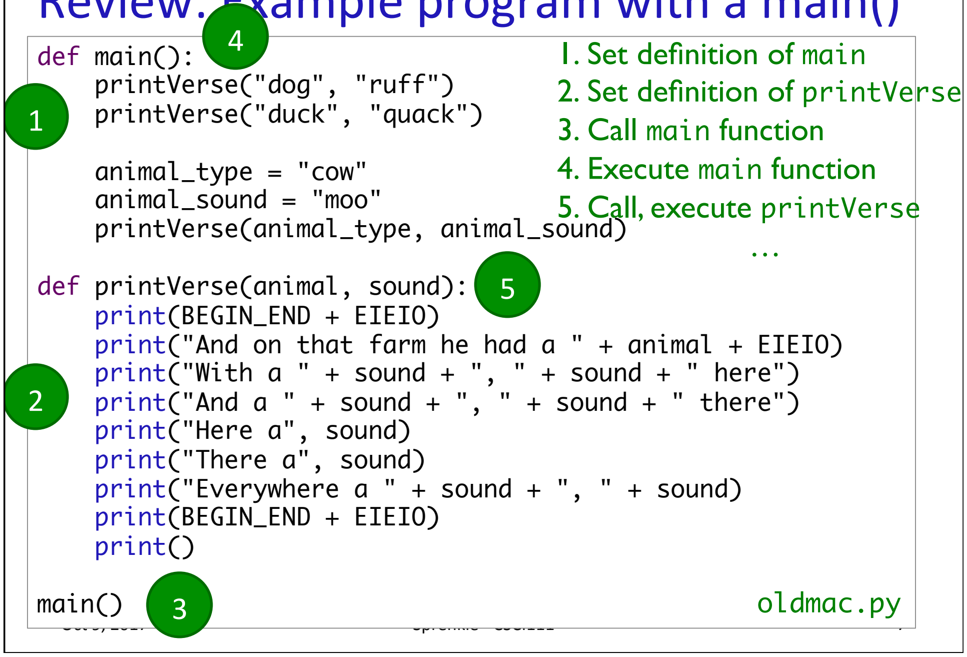
```
def main():
    printVerse("dog", "ruff")
    printVerse("duck", "quack")

    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

Review: 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

Review: Program Organization

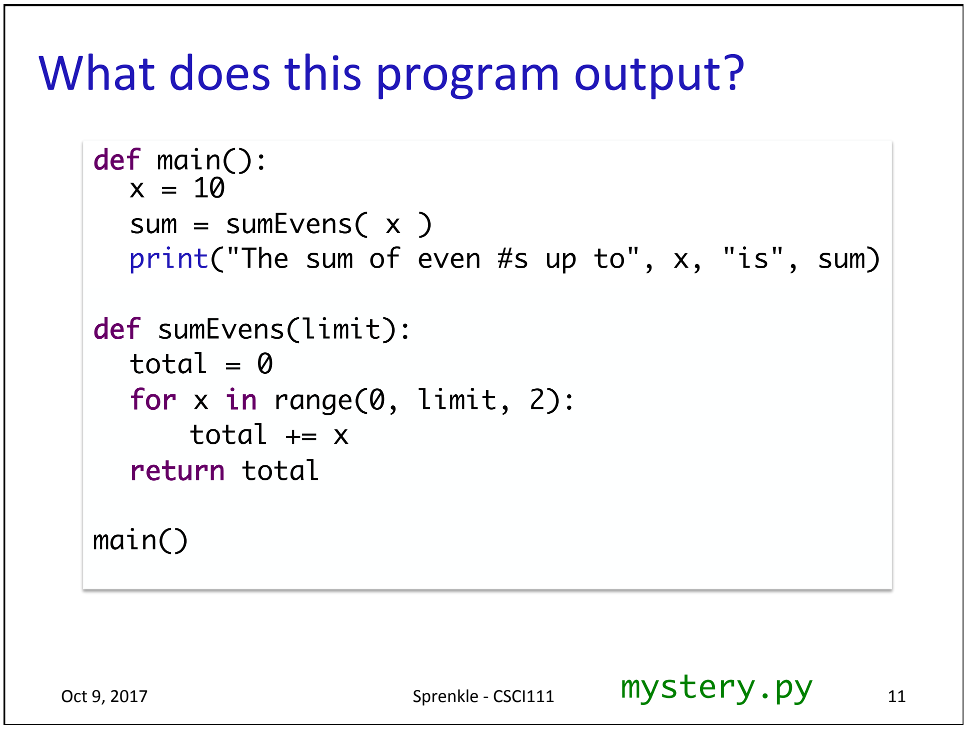
- Larger programs require **functions** to maintain readability
  - Use `main()` and other functions to break up program into *smaller*, more *manageable* chunks
  - "**Abstract** away" the details
- As before, can still write smaller scripts without any functions
  - Can try out functions using smaller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

Words in Different Contexts

> "Time flies like an arrow.
> Fruit flies like bananas."
> — Groucho Marx.

- Output from a **function**
  - What is returned from the function
  - If the function prints something, it's what the function *displays* (rather than outputs).
- Output from a **program**
  - What is displayed by the program

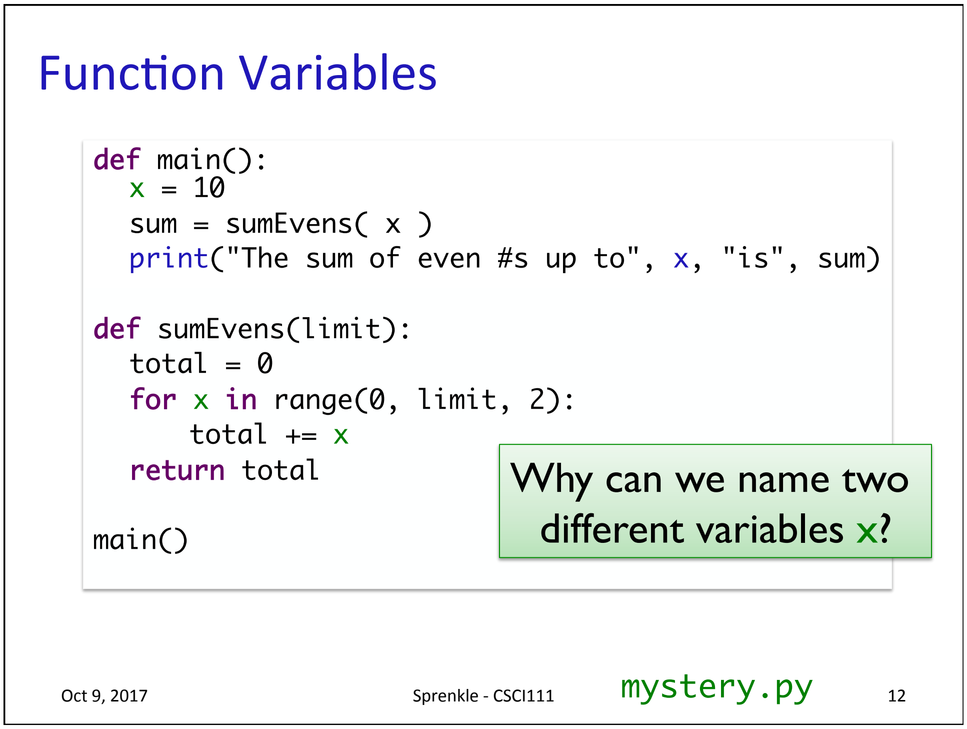
VARIABLE LIFETIMES AND SCOPE



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

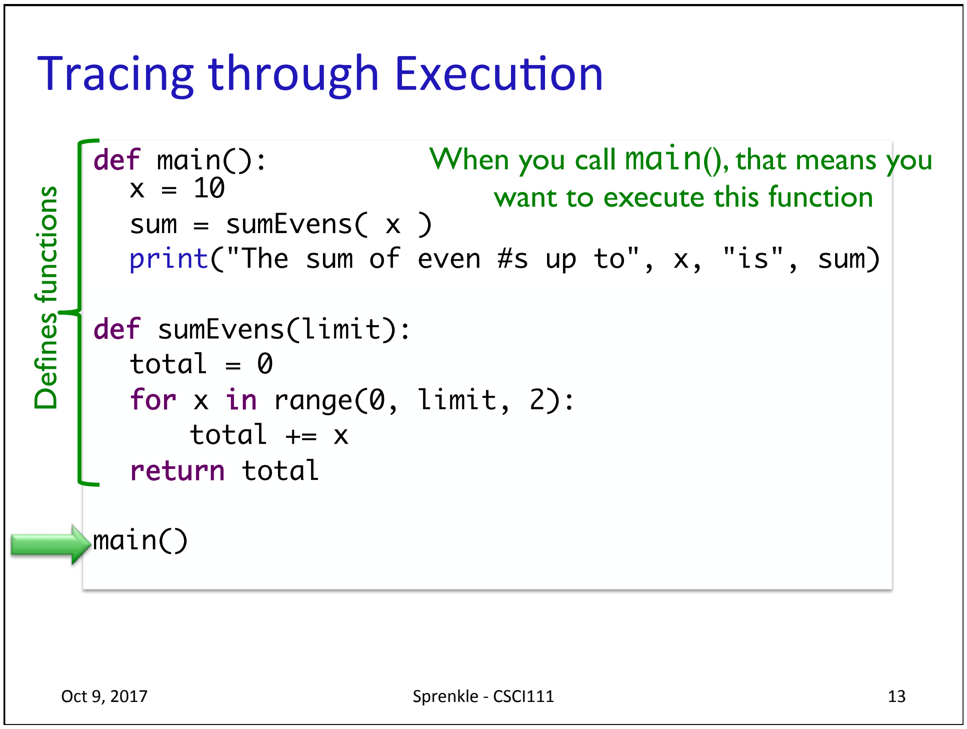


Function Variables

```
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

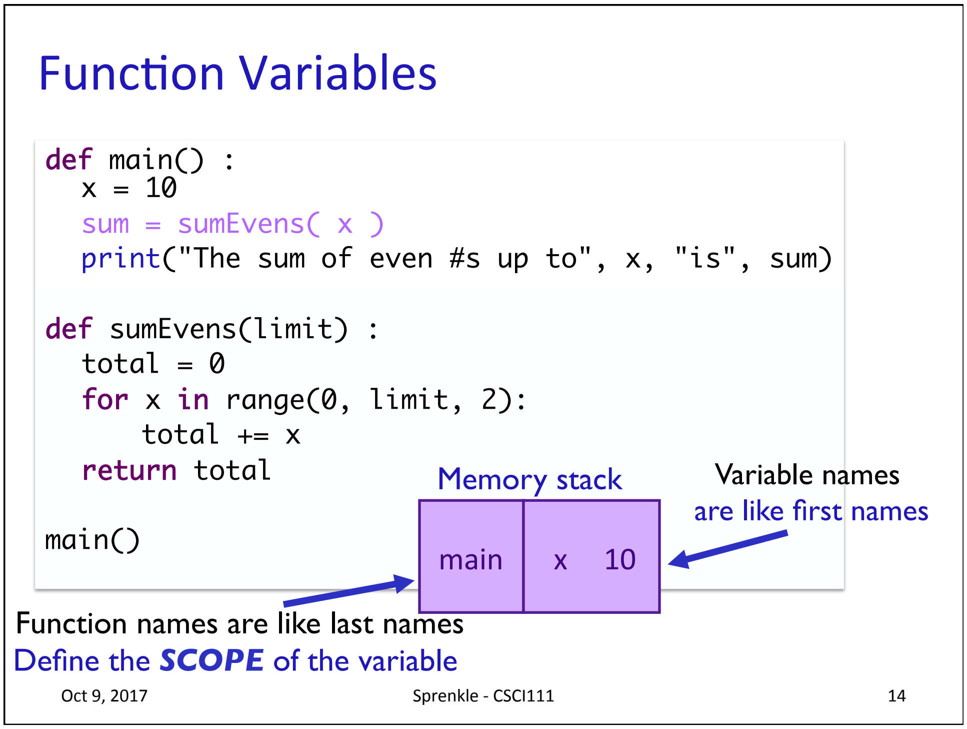


Tracing through Execution

Defines functions

```
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()
```

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()
```

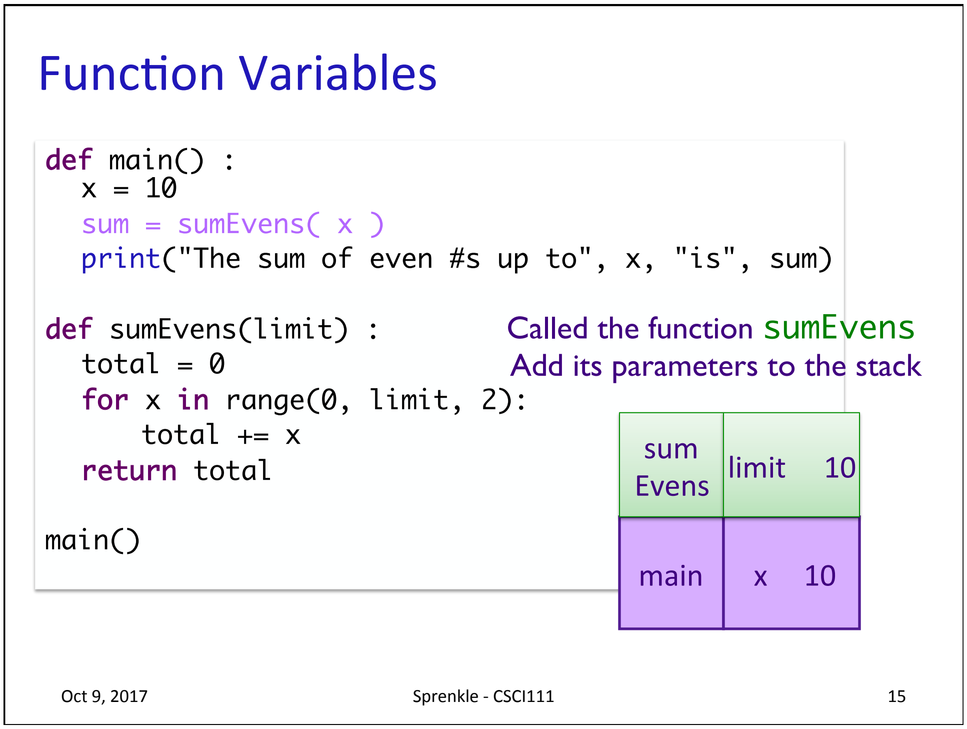
Memory stack

main	x 10

Variable names are like first names

Function names are like last names

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

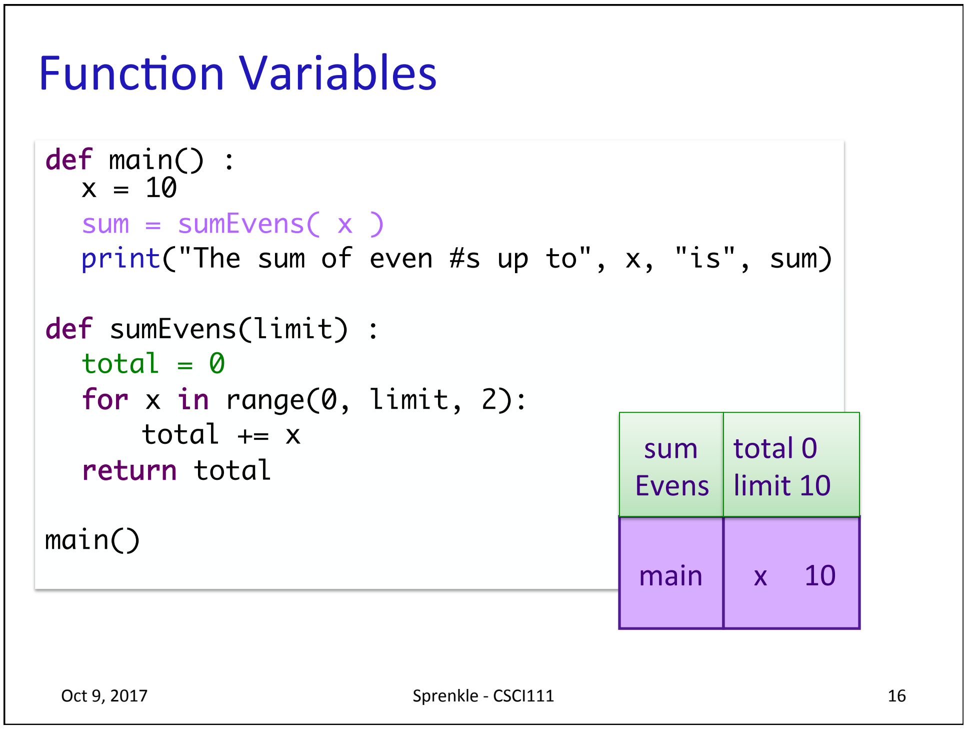


Function Variables

```
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



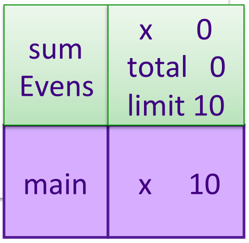
Function Variables

```
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

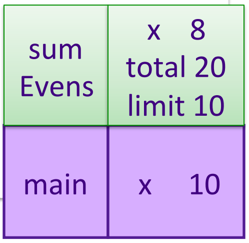
```
def main() :
   x = 10
   sum = sumEvens( x )
   print("The sum of even #s up to", x, "is", sum)

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()
```



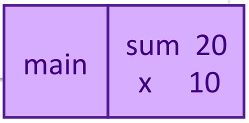
sum Evens	x 8 total 20 limit 10
main	x 10

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()
```

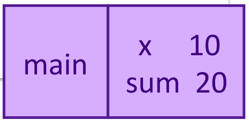
Function sumEvens returned

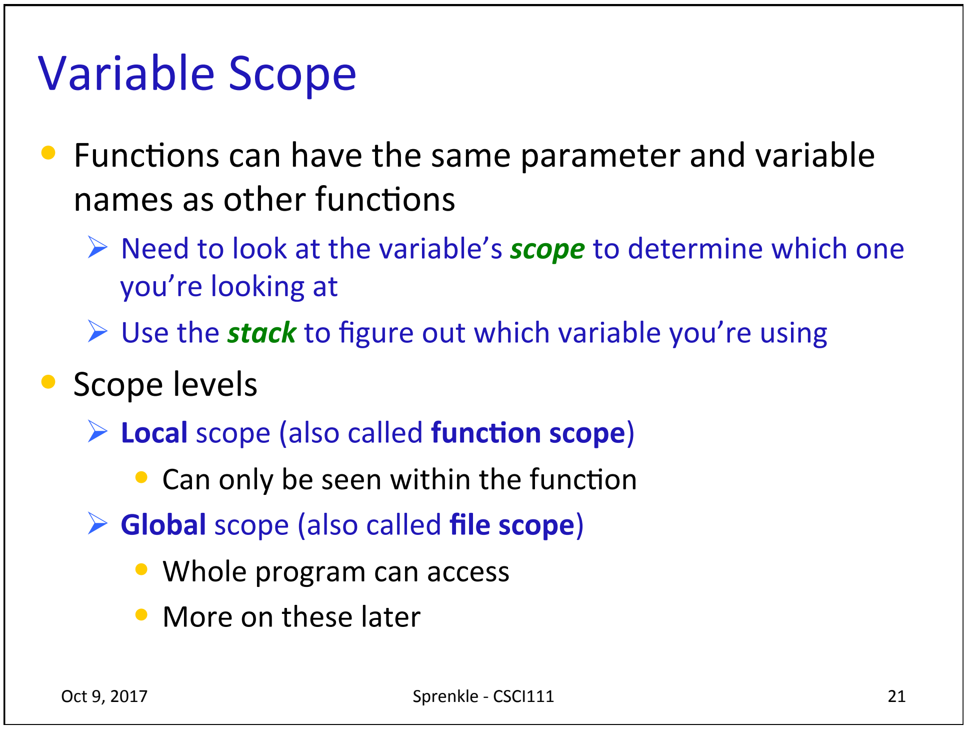
- no longer have to keep track of its variables on stack
- lifetime of those variables is over



Function Variables

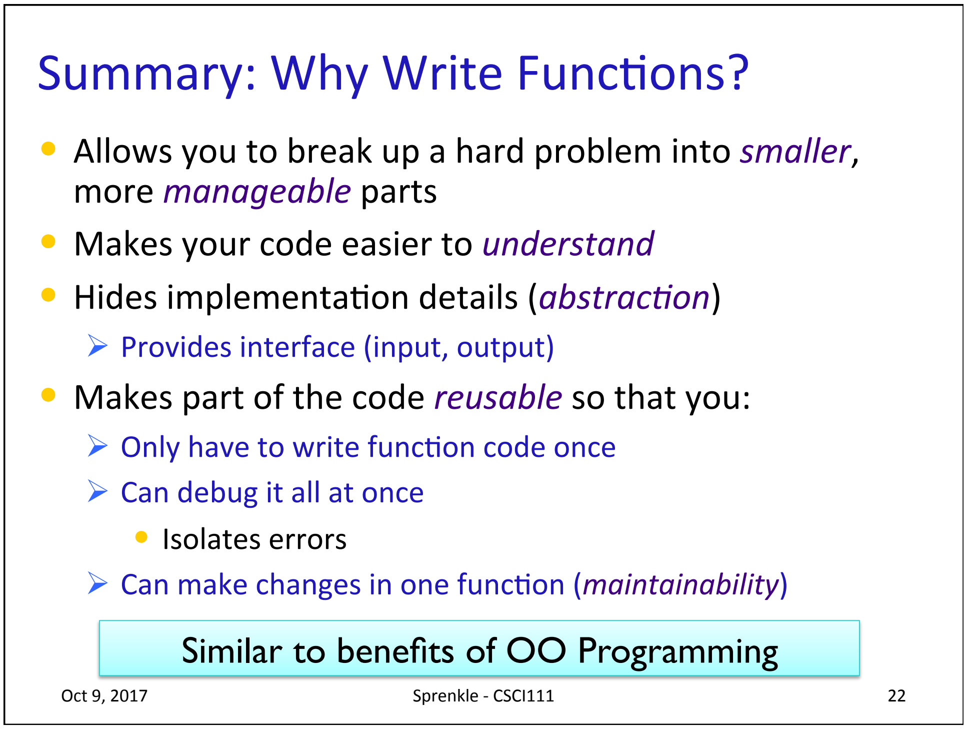
```
def main() :
   x = 10
   sum = sumEvens( x )
   print("The sum of even #s up to", x, "is", sum)

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

main()
```





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



Summary: Why Write Functions?

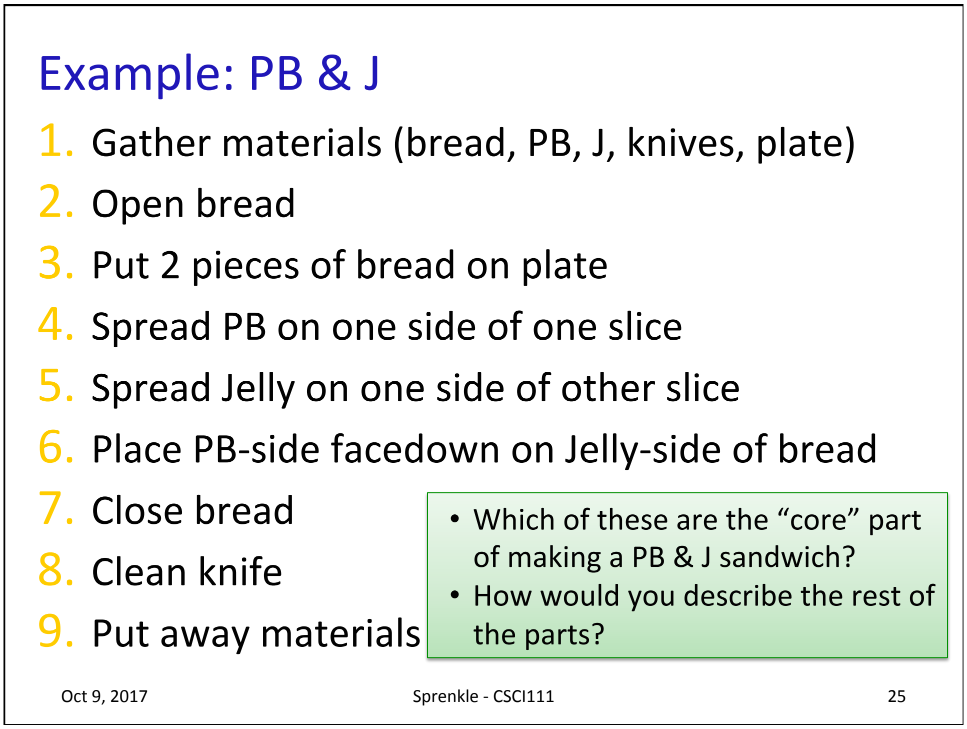
- Allows you to break up a hard problem into *smaller*, more *manageable* parts
- Makes your code easier to *understand*
- Hides implementation details (*abstraction*)
  - Provides interface (input, output)
- Makes part of the code *reusable* so that you:
  - Only have to write function code once
  -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

Similar to benefits of OO Programming

REFACTORING

Refactoring

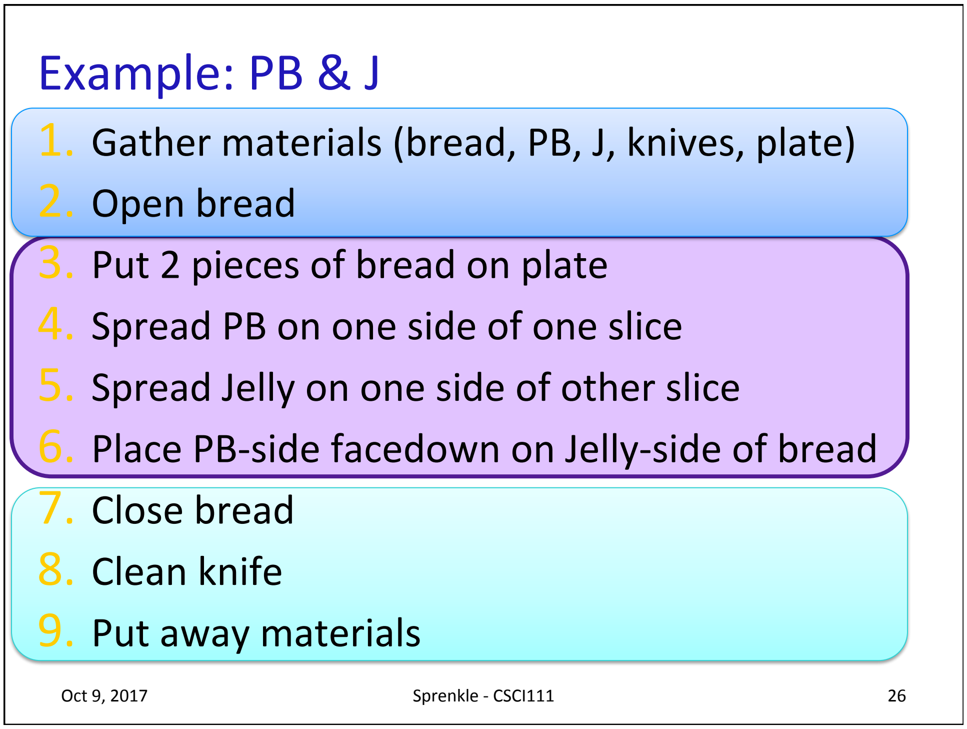
- After you’ve written some code and it passes all your test cases, the code is probably still not perfect
- ***Refactoring*** is the process of improving your code *without* changing its functionality
  - Organization
  - Abstraction
    - Example: Easier to read, change
  - Easier to test
- Part of iterative design/development process
- Where to refactor with functions
  - Duplicated code
    - “Code smell”
  - Reusable code
  - Multiple lines of code for one purpose



Example: PB & J

1. Gather materials (bread, PB, J, knives, plate)
2. Open bread
3. Put 2 pieces of bread on plate
4. Spread PB on one side of one slice
5. Spread Jelly on one side of other slice
6. Place PB-side facedown on Jelly-side of bread
7. Close bread
8. Clean knife
9. Put away materials

- Which of these are the “core” part of making a PB & J sandwich?
- How would you describe the rest of the parts?

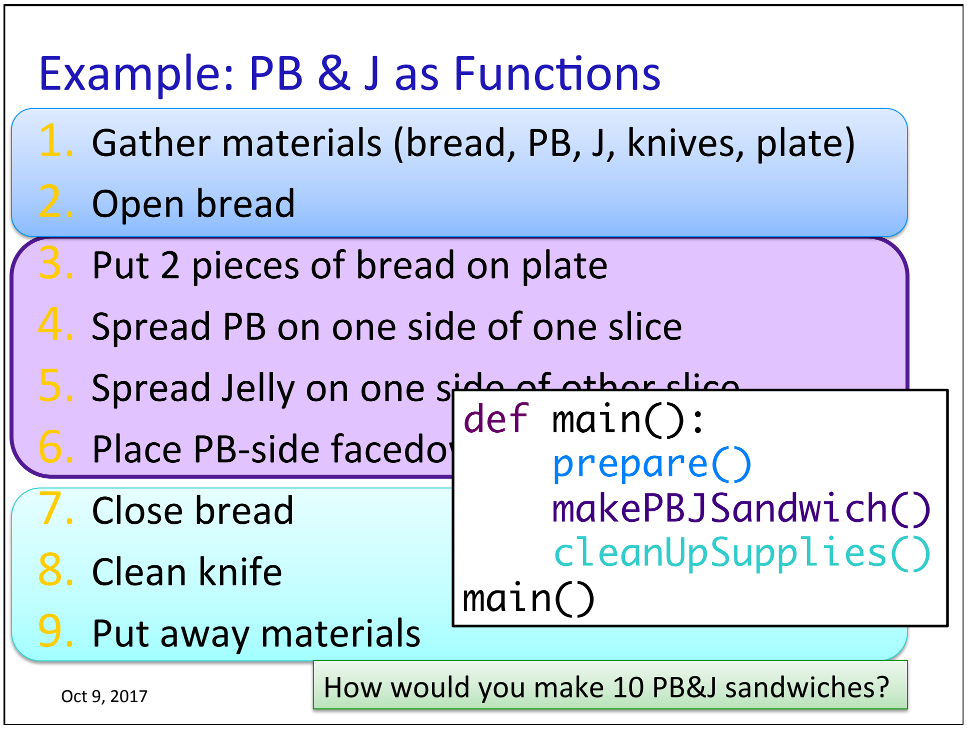


Example: PB & J

1. Gather materials (bread, PB, J, knives, plate)
2. Open bread

3. Put 2 pieces of bread on plate
4. Spread PB on one side of one slice
5. Spread Jelly on one side of other slice
6. Place PB-side facedown on Jelly-side of bread

7. Close bread
8. Clean knife
9. Put away materials



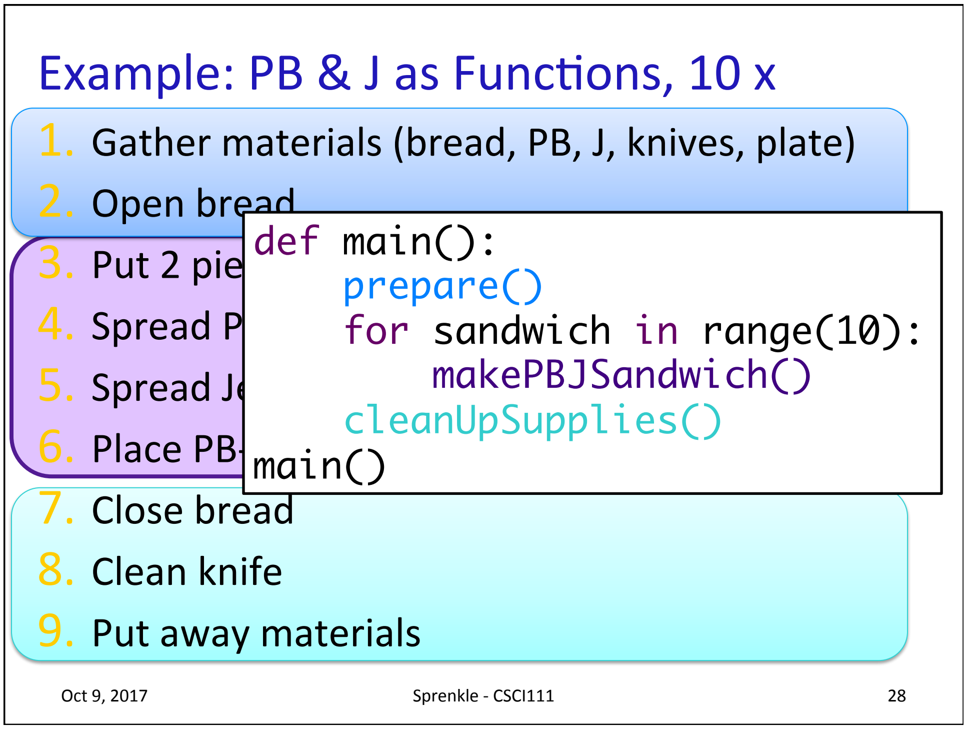
Example: PB & J as Functions

1. Gather materials (bread, PB, J, knives, plate)
2. Open bread
3. Put 2 pieces of bread on plate
4. Spread PB on one side of one slice
5. Spread Jelly on one side of other slice
6. Place PB-side facedo
7. Close bread
8. Clean knife
9. Put away materials

```
def main():
    prepare()
    makePBJSandwich()
    cleanUpSupplies()
main()
```

How would you make 10 PB&J sandwiches?

Oct 9, 2017

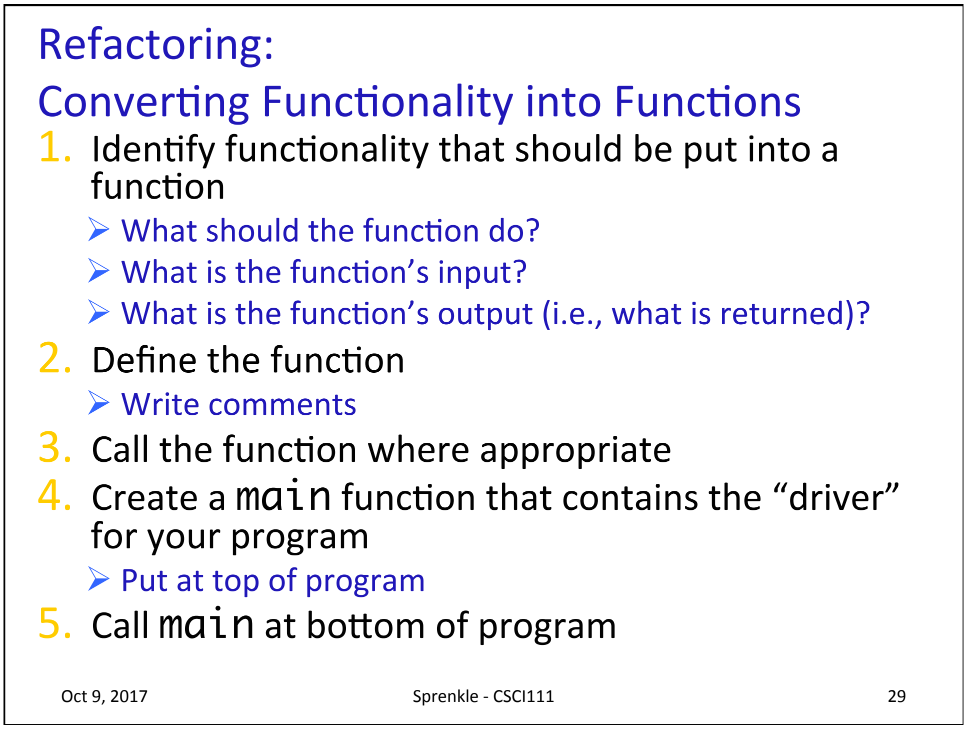


Example: PB & J as Functions, 10 x

1. Gather materials (bread, PB, J, knives, plate)
2. Open bread
3. Put 2 pie
4. Spread P
5. Spread J
6. Place PB-
7. Close bread
8. Clean knife
9. Put away materials

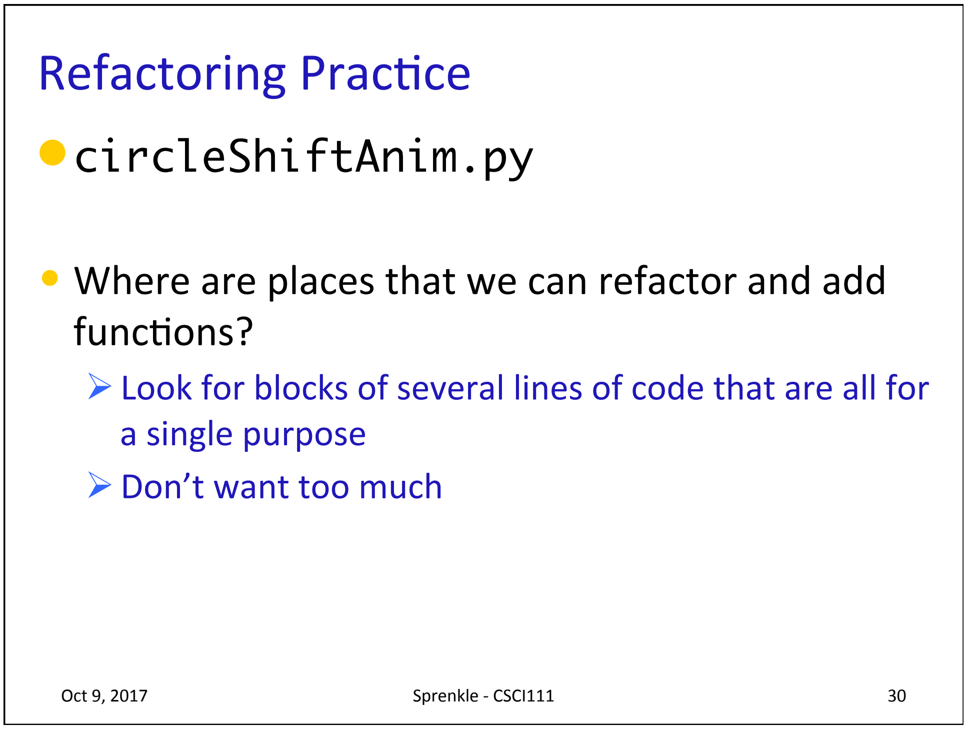
```
def main():
    prepare()
    for sandwich in range(10):
        makePBJSandwich()
    cleanUpSupplies()
main()
```

Oct 9, 2017 Sprenkle - CSCI111 28



Refactoring: Converting Functionality into Functions

1. Identify functionality that should be put into a function
   - ➢ What should the function do?
   - ➢ What is the function's input?
   - ➢ What is the function's output (i.e., what is returned)?
2. Define the function
   - ➢ Write comments
3. Call the function where appropriate
4. Create a `main` function that contains the "driver" for your program
   - ➢ Put at top of program
5. Call `main` at bottom of program

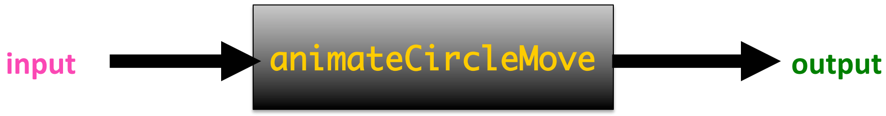


Refactoring Practice

- `circleShiftAnim.py`

- Where are places that we can refactor and add functions?
  - ➢ Look for blocks of several lines of code that are all for a single purpose
  - ➢ Don't want too much

Animate Circle Shift



input

animateCircleMove

output

- What it does: circle is animated, moving to a new position
- Input: circle, new center point
- Output: nothing returned

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!

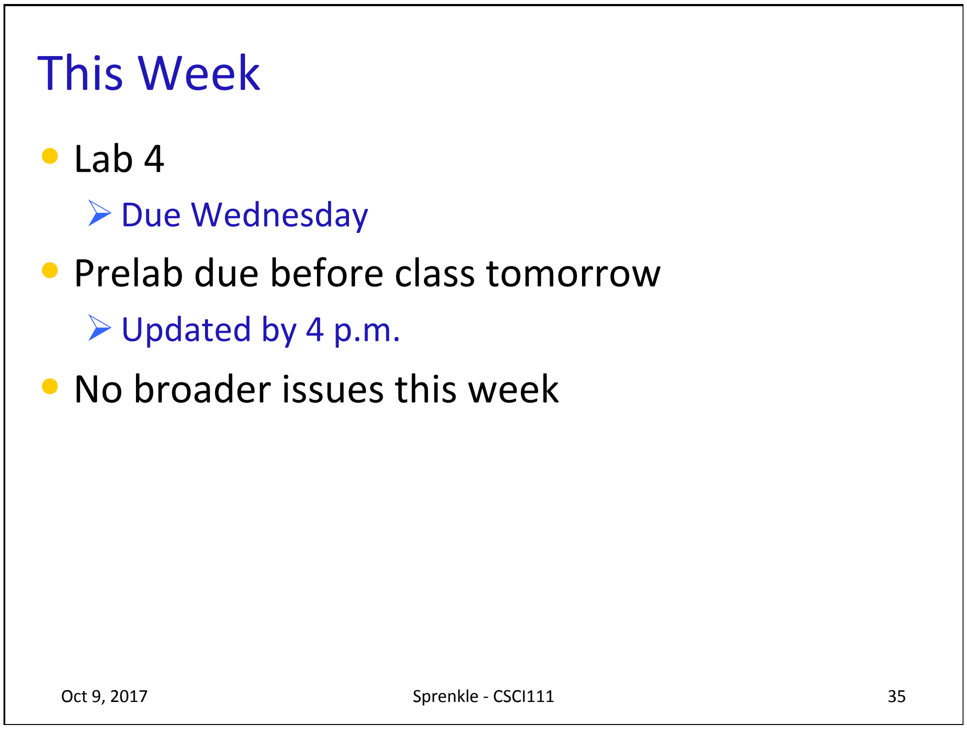
Example: Testing sumEvens

```
import test

def main():
    actual = sumEvens( 10 )
    expected = 20
    test.testEqual( actual, expected )

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

This is the actual result from our function

This is what we expect the result to be

testSumEvens.py



This Week

- Lab 4
  - Due Wednesday
- Prelab due before class tomorrow
  - Updated by 4 p.m.
- No broader issues this week