Objectives

- More on Functions
  - Scope, variable lifetime

Lab Review

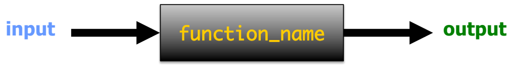
- Combinations of building block
- Faster recognition of how to solve
  - String without spaces → accumulate string!
- Iterative problem solving

Review: Functions

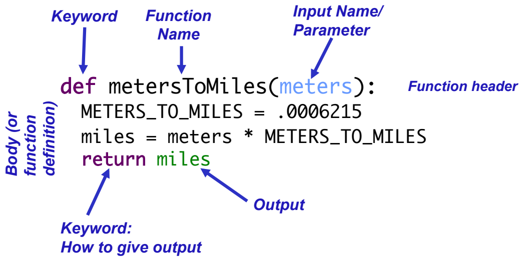
- What is the keyword to create a new function?
- What is the keyword to give output from a function?
- How do we give input to a function?
- Why write functions?

Review: Functions

- In general, a function can have
  - 0 or more inputs (the parameters)
  - 0 or 1 outputs (what is *returned*)
- When we define a function, we know its inputs and if it has output



Review: Syntax of Function Definition



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

Review: Parameters

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

Formal

Actual

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

Use: `roundCelc = round(celc, 2)`

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

Review: Function Output

- When the code reaches a statement like

  `return x`

  the function stops executing and

  x is the output ***returned*** to the place where 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 Input and Output

- What is the input and output to this function?

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

Function Input and Output

- 1 input: **meters**
- 1 output: the converted miles

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

Function Input and Output

- Identify 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

```
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

- Input? Output?

```
def printMenu():
    print "You have some options for what to do: "
    print "Enter an 'F' to find a song"
    print "Enter an 'S' to sort by Song title"
    print "Enter an 'A' to sort by Album"
    print "Enter an 'R' to sort by aRtist name"
    print "Enter an 'H' to list your options again"
    print "Enter a 'Q' to quit"
```

Function Input and Output

- 0 inputs and 0 outputs
  - Again, it *displays* something but does not `return` anything

```
def printMenu():
    print "You have some options for what to do: "
    print "Enter an 'F' to find a song"
    print "Enter an 'S' to sort by Song title"
    print "Enter an 'A' to sort by Album"
    print "Enter an 'R' to sort by aRtist name"
    print "Enter an 'H' to list your options again"
    print "Enter a 'Q' to quit"
```

Typical Refactoring Process

1. Identify functionality that should be put into a function
   - What is the function's input?
   - What is the function's output?
2. Define/write the function
   - Write descriptive 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

Converting functionality into functions

- `binaryToDecimal.py`
  - Converting from binary to decimal
  - *Checking if a string contains only binary numbers*

- Write comments for the functions

Review: Why write functions?

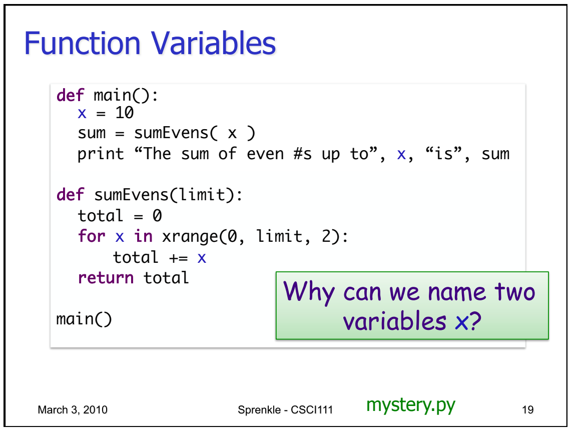
- Allows you to break up a hard problem into smaller, more manageable parts
- Makes your code easier to understand
- Hides implementation details (*abstraction*)
  - Provides interface (input, output)
- Makes part of the code reusable so that you:
  - Only have to write function code once
  - Can debug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

Similar to benefits of OO Programming

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 xrange(0, limit, 2):
      total += x
  return total

main()
```

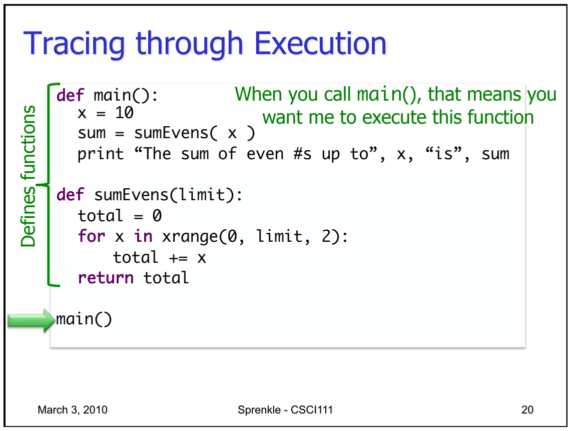


Function Variables

```
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 xrange(0, limit, 2):
      total += x
  return total

main()
```

Why can we name two variables x?

mystery.py

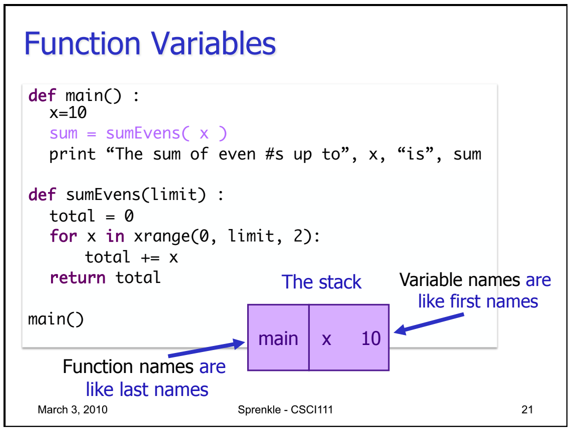


Tracing through Execution

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 xrange(0, limit, 2):
      total += x
  return total

main()
```

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



Function Variables

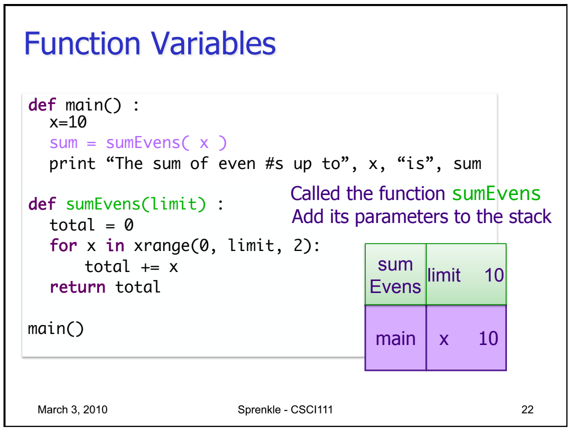
```
def main() :
  x=10
  sum = sumEvens( x )
  print “The sum of even #s up to”, x, “is”, sum

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

main()
```

The stack

main | x 10

Variable names are like first names

Function names are like last names

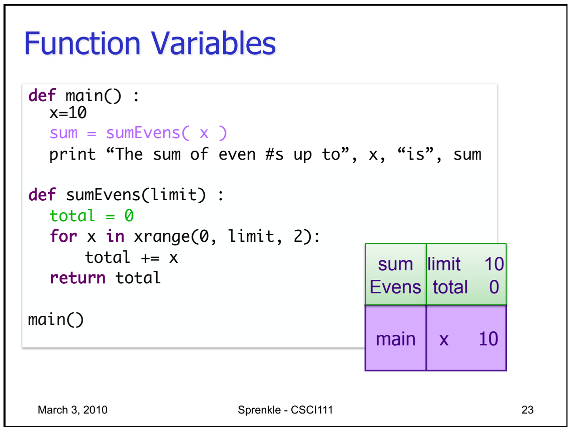


Function Variables

```
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 xrange(0, limit, 2):
      total += x
  return total

main()
```

Called the function sumEvens
Add its parameters to the stack

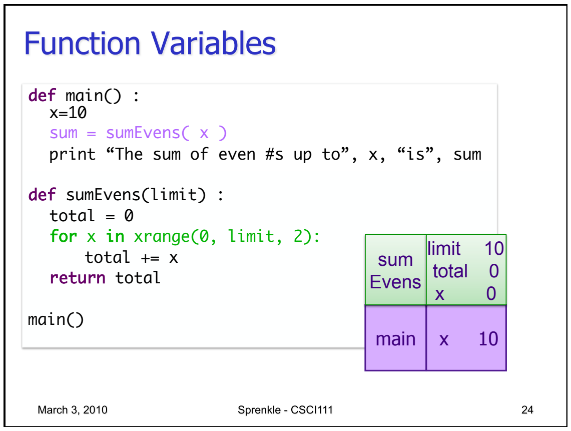
sum Evens | limit 10
main | x 10



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 xrange(0, limit, 2):
      total += x
  return total

main()
```

sum Evens | limit 10
total 0
main | x 10



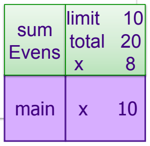
Function Variables

```
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 xrange(0, limit, 2):
      total += x
  return total

main()
```

sum Evens | limit 10
total 0
x 0
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 xrange(0, limit, 2):
     total += x
  return total

main()
```

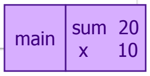


Function Variables

```
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 xrange(0, limit, 2):
     total += x
  return total

main()
```

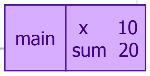
Function sumEvens returned

- 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 xrange(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

Function Scope

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

```
def main():
    binary_string = raw_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
```

Variable Scope

- Practice: `scope.py`
  - Trace through program--what does it do?
- Answer questions in program…

Practice

- What is the output of this program?
  - Example: user enters 4

```
def main():
    num = input("Enter a number to be squared: ")
    square = square(num)
    print "The square is:", square

def square(n):
    return n * n

main()
```

practice1.py

Practice

- What is the output of this program?
  - Example: user enters 4

```
def main():
    num = input("Enter a number to be squared: ")
    squared = square(num)
    print "The square is:", squared
    print “The original num was:”, n

def square(n):
    return n * n

main()
```

practice2.py

Practice

- What is the output of this program?
  - Example: user enters 4

```
def main():
    num = input("Enter a number to be squared: ")
    squared = square(num)
    print "The square is:", squared
    print “The original num was:”, n

def square(n):
    return n * n

main()
```

Error! **n** does not have a value in function `main()`

Variable Scope

- Know “lifetime” of variable
  - Only during execution of function
  - Related to idea of “scope”
- What about variables outside of functions?
  - Example: non_function_vars.py

Why We Don’t Want Variables to Have Global Scope

- Other functions modify our data
  - Unintentionally from our point of view …

Passing Parameters

- Only **copies** of the actual parameters are given to the function
  - For **immutable** data types (which are what we’ve talked about so far)
- The *actual* parameters in the calling code do not change
- **Swap example:**
  - Swap two values in script
  - Then, put into a function

```
x = 5
y = 7
```

→

```
x = 7
y = 5
```

WHAT MAKES A GOOD FUNCTION?

Writing a “good” function

- Should be an “intuitive chunk”
  - Doesn’t do too much or too little
- Should be reusable
- Always have comment that tells what the function does

Good vs. Bad Functions

- Bad: Does too little

```
def getUserInput():
     input = input(“Enter a number”)
     return input
```

- Good: Validates the input

```
def getUserInput():
     input = input(“Enter a number”)
     while input <= 0:
          print “Number must be positive”
          input = input(“Enter a number”)
     return input
```

Debugging Advice

- Build up your program in steps
  - Always write only small pieces of code
  - Test, debug. **Repeat**
- Write function body as part of `main`, test
  - Then, separate out into its own function
  - Similar to process using in lab probs
- Test function separately from other code
  - Comment out irrelevant code to make sure that the function behaves as expected

---

This Week

- Lab 6 due Friday
- Broader Issue: Volunteer Computing

- Reminder: Next Friday, Mar 12, Exam