STRUCTURING PROGRAMS: FUNCTIONS

ES 112

Brief Recap

- Understanding Functions and Function Invocations

Menu for Today!

- Induction and recursion
- More on scope
- Passing parameters
- Example: Palindrome

Induction and Recursion

- Proof By Induction

$$\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$$

- How would you prove this?

- Definition of Factorial

$$n! = n \times (n-1) \times \cdots \times 2 \times 1$$

- This is not a very nice definition: what does ... mean?
- A cleaner way to define factorial:

$$0! = 1$$
$$n! = n \times (n-1)!$$

- This is known as Recursion

Recursion and Induction are closely linked

Examples of Recursion in the Real World

- Language grammars are recursive:
- Noun phrase

  NP := NP PP

  NP := (Det) (Adj) NP

  PP := Prep NP

- Covid-19 Contact tracing is recursive
  - *For each patient, identify all contacts*
  - *Test all contacts*
  - *For each contact that tests positive, repeat contact tracing*
  - *If no contacts test positive, terminate*

Computing Factorials!

```
def factorialI(n):
    result = 1
    for i in range(1,n+1):
        result = result * i
    return(result)

def factorialR(n):
    if (n == 0)
        return(1)
    else
        return (n * factorial(n - 1))
```

Iteration and recursion are equivalent

Recursive Programs Can Sometimes be Simpler To Understand

```
def fibR(n):
    if n == 0 or n == 1:
        return 1
    else:
        return fibR(n-1) + fibR(n-2)
def fibI(n):
    oldFib = 1
    newFib = 1
    for i in range(1,n):
        oldFib, newFib = newFib, (oldFib + newFib)
    return newFib
```

Computing Factorials!
Handling Errors

```
def factorial(n):
    if (n < 0):
        print('Cannot compute factorial for a negative number')
        return None
    elif (n == 0)
        return(1)
    else
        return (n * factorial(n - 1))
```

This is messy: we are doing error checking n times
We should ideally do it only once

Computing Factorials: Nested Functions

```
def factorial(n):
    def innerFactorial(n):
        if (n == 0):
            return(1)
        else
            return n * innerFactorial(n-1)

    if (n < 0):
        print('Cannot compute factorial for a negative number')
        return None
    else
        return innerFactorial(n)
```

Notice : we defined a function inside a function!

Functions Hide Away Messy Details

```
def circumference(radius):
    pi = 3.1416
    return(2*pi*radius)
```

- Value of pi assigned in the function is not accessible outside the function definition
- This allows us to create temporary variables and use them for our calculations
- The user of the function does not need to know about these variables

Scope!

- A variable binding created in a function is not available for use when you return from a function
- In the main program, you cannot use a variable before you assign it an value
- In a function, you can use (read) a variable even if you have not assigned it an value **in the function**
  - *You must assign it a value somewhere though, before you can use it*

How do we figure out what value a variable in a function takes if we have not assigned it a value in the function

Accessing a Variable Not Assigned in the Function

```
pi = 3.14
def circumference(radius):
    print(2*pi*radius)
def area(radius):
    pi = 3.1416
    print(pi*radius**2)
```

- In the function, it appears as though we are using `pi` before assigning a value to it
- In this case, `pi` take the value assigned to it in the main program
- This allows us to declare a variable once, and use it across multiple functions
- If we change the value of `pi` inside the function, the changed value is not visible outside the function

Global Variables

- Looking up a variable binding in a function,
  - *first check if the variable is defined in the function*
  - *if it isn't, check if the variable is defined in the main program*
- Variables that are assigned values in the main program can be read anywhere in the program (including inside functions)
- However, if we try to assign values inside a function to a variable that is defined in the main program, a new variable with the same name will be created inside the function
- If we want to change the value of variable assigned in the main program, we must declare the variable as global in the function

```
pi = 3.14
def area(radius):
    global pi
    pi = 3.1416
    print(pi*radius**2)
```

What About Variables in Nested Functions

```
def outer_function():
    superhero = "I am Batman!"
    print(f"Outer function says, {superhero}")
    def inner_function():
        superhero += " I am kidding!!"
        print(f"Inner function says, {superhero}")
    inner_function()
    print(f”After return from Inner function, {superhero}”)
outer_function()
```

Looking Up Variables in Nested Functions

- Looking up a variable binding in a function,
  - *first check if the variable is defined in the function*
  - *if it isn't, check if the variable is defined in the main program*
- This rule does not include looking up the function binding in an enclosing function
- If you want to look up bindings declared in enclosing functions, use the keyword `nonlocal`
- `nonlocal` will look up not only the immediate enclosing function but any function that encloses the current function.
  - *It will not look in the global scope though*
- Note that we are looking for variables in the functions that enclose the definition of the current function; not in the context of the function that calls the current function

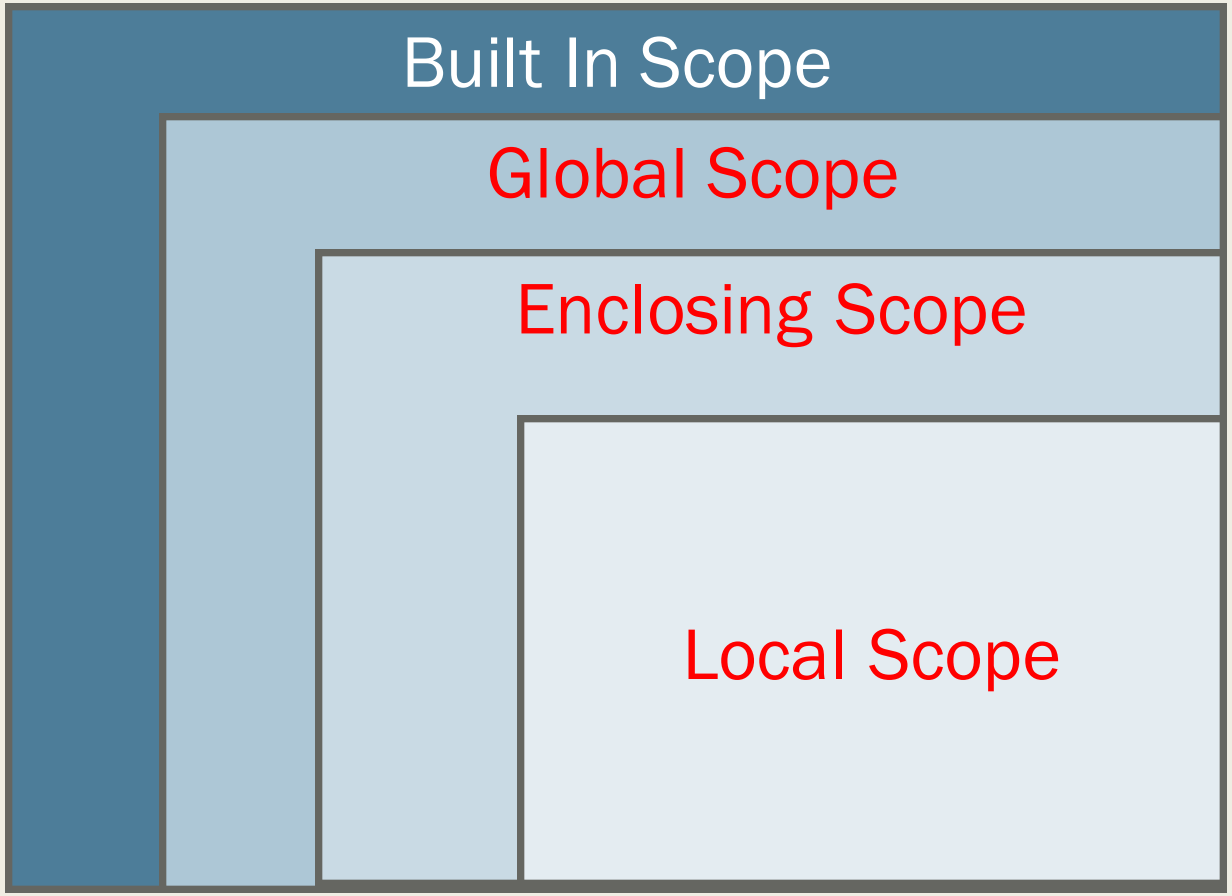
Static and Dynamic Scope

```
def f():
    x = 1
    def g():
        nonlocal x
        print(f'In g, x is {x}')
    def h():
        x = 2
        g()
    h()

x = 4
f()
```

Should `x` take the value defined in `main`, in `f` or in `h`?

Different Kinds of Scope in Python



Summarizing Scope

- A variable is only available from inside the region it is created.
- Allocation
  - *Variables defined in* `main` *are allocated in* `global` *scope, and are accessible from anywhere*
    - In order to change the value of a variable allocated in global scope from within a function, use the keyword `global`
  - *Variables defined in a function* `f` *are allocated in the scope of the function* `f` *and are accessible within the function, and in all functions defined inside* `f`
    - If the same name is used for variable inside and outside a function, you talking about two different variables
    - In order to change the value of a variable allocated in an enclosing scope from within a function, use the keyword `nonlocal`
    - `nonlocal` variables are assigned bindings based on scope enclosing the function textually, not dynamically

Keyword Arguments

- Usually, arguments are matched to parameters in the sequence in which they are defined

```
def divide(num, den):
    return(num / den)
divide(2,3)
```

- `num` is bound to `2`, `den` is bound to `3`
- We can also provide arguments in a different order by specifying which parameter they should be bound to
- `divide(den = 3, num = 2)` is equivalent to `divide(2,3)`

Default Values for Parameters

- We can specify default values for parameters in the function definition

```
def growth(intRate, period = 1):
    return (1 + intRate)**period

todaysRate = 0.07
```

`growth(todaysRate, 5)` evaluates to 1.4025517307000004

`growth(todaysRate)` evaluates to `1.07`

Palindrome

- A number is a palindrome if

  `number == reverse(number)`
- Three approaches to computing reverse
  - Iteration
  - Recursion using a global variable to store the result
  - Recursion without global variables
- reverse(12345) -> 54321
  - Method 1:
    reverse(12345) = 54321 = ((((5*10 + 4)*10 + 3)*10) + 2)*10 + 1
  - Method 2:
    reverse(12345) = 54321 = $5*10^4$ + reverse(1234)

Compute Reverse By Iteration

```
def reverse(number):
    result = 0
    while (number >= 1):
        result = result*10 +
                 number % 10
        number = number // 10
    return result
```

Compute Reverse : Method 1 with Recursion

```
def reverse1(number):
  global result
  result = 0
  innerRev1(number)

  def innerRev1(number):
    global result
    if (number >= 1):
      result = result *10 +
                 number % 10
      innerRev1(number // 10)
    return
```

- This is exactly the same logic as the iterative method
  - *The last line of innerRev1 is the recursive call: **Tail Recursion***
- No return value needed as the result of the computation is stored in a global variable
  - *Side effects*
- Difficult to understand all the lines of code that affect the value of result
  - *Difficult to debug*

Compute Reverse : Method 2 with Recursion

```
def numDig(number):
  if number < 10:
    return 1
  else:
    return 1 +
       numDig(number // 10)

def reverse2(number):
    numDigits = numDig(number)
    return innerRev2(number,
                     numDigits - 1)
```

```
def innerRev2(number, power):
    if power == 0:
        return number
    else:
        lastDigit = number % 10
        remaining = number // 10
        return
          (lastDigit*(10**power) +
           innerRev2(remaining, power - 1)
```

Compute Reverse : Method 3 with Recursion

```
def reverse3(number):
    return innerRev3(number,0)


def innerRev3(number, result):
    if (number >= 1):
        newResult = result *10 + number % 10
        return innerRev3(number // 10, newResult)
    else:
        return result
```