

Variable Scope

SECTION 5.8

Variable Scope

- Variables can be declared:
 - Inside a function
 - Known as 'local variables'
 - Only available inside this function
 - Parameter variables are like local variables
 - Outside of a function
 - Sometimes called 'global scope'
 - Can be used (and changed) by code in any function
- How do you choose?

The scope of a variable is the part of the program in which it is visible

Examples of Scope

- `sum`, `square` & `i` are local variables in main

```
def main() :  
    sum = 0  
    for i in range(11) :  
        square = i * i  
        sum = sum + square  
    print(square, sum)
```

The diagram illustrates the scope of variables in the code. A green bracket groups the lines `square = i * i` and `sum = sum + square`, with the label `square` below it. A red bracket groups the lines `for i in range(11) :`, `square = i * i`, and `sum = sum + square`, with the label `i` to its right. A blue bracket groups the lines `sum = 0`, `sum = sum + square`, and `print(square, sum)`, with the label `sum` to its right.

Local Variables of functions

- Variables declared inside one function are not visible to other functions
 - `sideLength` is local to main
 - Using it outside main will cause a compiler error

```
def main():
    sideLength = 10
    result = cubeVolume()
    print(result)

def cubeVolume():
    return sideLength * sideLength * sideLength # ERROR
```

Re-using Names for Local Variables

- Variables declared inside one function are not visible to other functions
 - `result` is local to `square` and `result` is local to `main`
 - They are two different variables and do not overlap
 - This can be very confusing

```
def square(n):  
    result = n * n  
    return result  
    } result  
  
def main():  
    result = square(3) + square(4)  
    print(result)  
    } result
```

Global Variables

- They are variables that are defined outside functions
- A global variable is visible to all functions that are defined after it
- However, any function that wishes to use a global variable must include a global declaration

Example Use of a Global Variable

- If you omit the global declaration, then the balance variable inside the withdraw function is considered a local variable

```
balance = 10000    # A global variable
def withdraw(amount) :
    # This function intends to access the
    # global 'balance' variable
    global balance
    if balance >= amount :
        balance = balance - amount
```

Programming Tip

- There are a few cases where global variables are required (such as `pi` defined in the `math` module), but they are quite rare
- Programs with global variables are difficult to maintain and extend because you can no longer view each function as a “black box” that simply receives arguments and returns a result
- Instead of using global variables, use function parameter variables and return values to transfer information from one part of a program to another

Recursive Functions

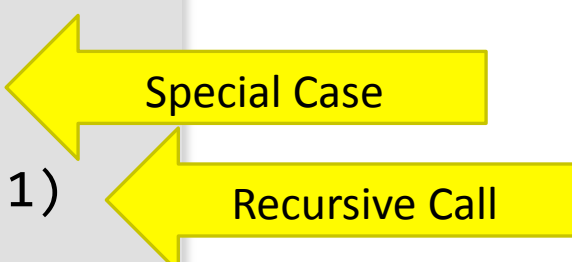
SECTION 5.10

Recursive Functions

- A recursive function is a function that calls itself
- A recursive computation solves a problem by using the solution of the same problem with simpler inputs
- For a recursion to terminate, there must be special cases for the simplest inputs

Recursive Triangle Example

```
def printTriangle(sideLength) :  
    if sideLength < 1 : return  
    printTriangle(sideLength - 1)  
    print("[]" * sideLength)
```



```
[]  
>[]  
>[] []  
>[] [] []
```

Print the triangle with side length 3.

Print a line with four [].

- The function will call itself (and not output anything) until sideLength becomes < 1
- It will then use the return statement and each of the previous iterations will print their results
 - 1, 2, 3 then 4

Recursive Calls and Returns

Here is what happens when we print a triangle with side length 4.

- The call `printTriangle(4)` calls `printTriangle(3)`.
 - The call `printTriangle(3)` calls `printTriangle(2)`.
 - The call `printTriangle(2)` calls `printTriangle(1)`.
 - The call `printTriangle(1)` calls `printTriangle(0)`.
 - The call `printTriangle(0)` returns, doing nothing.
 - The call `printTriangle(1)` prints `[]`.
 - The call `printTriangle(2)` prints `[][]`.
 - The call `printTriangle(3)` prints `[][][]`.
 - The call `printTriangle(4)` prints `[][][][]`.

A Second Example

- Open the file `digits.py`
- This program computes the sum of the digits in a number (n)
 - We solved this last chapter in Section 4.2
 - We will use $n = 1729$ as our example
- Our algorithm was:
 - Remove the last digit by computing $n // 10$ and add the remainder to our total
 - To use recursion we can use the recursive function:
 - $\text{digitsum}(n // 10) + n \% 10$
 - Our special case is $n == 0$ to terminate the recursion

Summary

Summary: Functions

- A function is a named sequence of instructions
- Arguments are supplied when a function is called
- The return value is the result that the function computes
- When declaring a function, you provide a name for the function and a variable for each argument
- Function comments explain the purpose of the function, the meaning of the parameters and return value, as well as any special requirements
- Parameter variables hold the arguments supplied in the function call

Summary: Function Returns

- The **return** statement terminates a function call and yields the function result
 - Complete computations that can be reused into functions
- Use the process of stepwise refinement to decompose complex tasks into simpler ones
 - When you discover that you need a function, write a description of the parameter variables and return values
 - A function may require simpler functions to carry out its work

Summary: Scope

- The scope of a variable is the part of the program in which the variable is visible
 - Two local or parameter variables can have the same name, provided that their scopes do not overlap
 - You can use the same variable name within different functions since their scope does not overlap
 - Local variables declared inside one function are not visible to code inside other functions

Summary: Recursion

- A recursive computation solves a problem by using the solution of the same problem with simpler inputs
 - For recursion to terminate, there must be special cases for the simplest inputs
 - The key to finding a recursive solution is reducing the input to a simpler input for the same problem
 - When designing a recursive solution, do not worry about multiple nested calls
 - Simply focus on reducing a problem to a slightly simpler one