Lecture 4: Functions

Course outline

- Part 1 Introduction to Computing and Programming (first 2 weeks):
  - Problem solving: Problem statement, algorithm design, programming, testing, debugging
  - Scalar data types: integers, floating point, Boolean, others (letters, colours)
  - Arithmetic, relational, and logical operators, and expressions
  - Data representation of integers, floating point, Boolean
  - Composite data structures: string, tuple, list, dictionary, array
  - Sample operations on string, tuple, list, dictionary, array
  - Algorithms (written in pseudo code) vs. programs
  - Variables and constants (literals): association of names with data objects
  - A language to write pseudo code
  - Programming languages: compiled vs. interpreted programming languages
  - Python as a programming language
  - Computer organization: processor, volatile and non-volatile memory, I/O

Course outline (may change a bit)

- Part 2 Algorithm design and Programming in Python (balance 11 weeks):
  - Arithmetic/Logical/Boolean expressions and their evaluations in Python
  - Input/output statements (pseudo code, and in Python)
  - Assignment statement (pseudo code, and in Python)
  - Conditional statements, with sample applications
  - Iterative statements, with sample applications
  - Function sub-programs, arguments and scope of variables
  - Recursion
  - Modules
  - Specific data structures in Python (string, tuple, list, dictionary, array), with sample applications
  - Searching and sorting through arrays or lists
  - Handling exceptions
  - Classes, and object-oriented programming
  - (Time permitting) numerical methods: Newton Raphson, integration, vectors/matrices operations, continuous-time and discrete-event simulation

Functions, or sub-programs

- You may define your own functions (also called sub-programs) similar to those you have already come across, such as print(), input(), float(), etc.
  - You may use these functions as if these were statements, but with different arguments, or parameters
    - print('hello x + z = ', x+z)
    - x = float(z)
    - k-range = range(1, 101, 1)
    - uv-product = dot-product(n, u, v)
  - Advantages of user defined functions
    - Reduce code duplication
    - Clarity of the code can be improved
    - Information hiding
    - Code reuse
    - Complex problem can be decomposed into simpler pieces

Functions, or sub-programs

- You may define your own functions (also called sub-programs) similar to those you have already come across, such as print(), input(), float(), etc.
  - You may use these functions as if these were statements, but with different arguments, or parameters
    - print('hello x + z = ', x+z)
    - x = float(z)
    - k-range = range(1, 101, 1)
    - uv-product = dot-product(n, u, v)
  - Advantages of user defined functions
    - Reduce code duplication
    - Clarity of the code can be improved
    - Information hiding
    - Code reuse
    - Complex problem can be decomposed into simpler pieces

Functions, or sub-programs

- You may define your own functions (also called sub-programs) similar to those you have already come across, such as print(), input(), float(), etc.
  - You may use these functions as if these were statements, but with different arguments, or parameters
    - print('hello x + z = ', y+z)
    - x = float(z)
    - k-range = range(1, 101, 1)
    - uv-product = dot-product(n, u, v)
  - Advantages of user defined functions
    - Reduce code duplication
    - Clarity of the code can be improved
    - Information hiding
    - Code reuse
    - Complex problem can be decomposed into simpler pieces

Functions, or sub-programs

- Consider application that computes something repeatedly but with different data
- Example: compute carper area of a flat:

```
# compute "carpet area" of 1BHK flat
carpet_area = 0
#Drawing+dining
x = float(input('x of DD_room '))
y = float(input('y of DD_room '))
carpet_area = carpet_area + x*y
#Bedroom
x = float(input('x of Bedroom '))
y = float(input('y of Bedroom '))
carpet_area = carpet_area + x*y
#Kitchen
x = float(input('x of Kitchen '))
y = float(input('y of Kitchen '))
carpet_area = carpet_area + x*y
#Bath
x = float(input('x of Bath '))
y = float(input('y of Bath '))
carpet_area = carpet_area + x*y
#Balcony
x = float(input('x of Balcony '))
y = float(input('y of Balcony '))
carpet_area = carpet_area + x*y
print('carpet area of 1 BHK flat', carpet_area)
```

Functions, or sub-programs

```
# compute "carpet area" of 1BHK flat
carpet_area = 0
#Drawing+dining
x = float(input('x of DD_room '))
y = float(input('y of DD_room '))
carpet_area = carpet_area + x*y
#Bedroom
x = float(input('x of Bedroom '))
y = float(input('y of Bedroom '))
carpet_area = carpet_area + x*y
#Kitchen
x = float(input('x of Kitchen '))
y = float(input('y of Kitchen '))
carpet_area = carpet_area + x*y
#Bath
x = float(input('x of Bath '))
y = float(input('y of Bath '))
carpet_area = carpet_area + x*y
#Balcony
x = float(input('x of Balcony '))
y = float(input('y of Balcony '))
carpet_area = carpet_area + x*y
print('carpet area of 1 BHK flat', carpet_area)
```

Functions, or sub-programs

```
# compute "carpet area" of 1BHK flat
carpet_area = 0
#Drawing+dining
x = float(input('x of DD_room '))
y = float(input('y of DD_room '))
carpet_area = carpet_area + x*y
#Bedroom
x = float(input('x of Bedroom '))
y = float(input('y of Bedroom '))
carpet_area = carpet_area + x*y
#Kitchen
x = float(input('x of Kitchen '))
y = float(input('y of Kitchen '))
carpet_area = carpet_area + x*y
#Bath
x = float(input('x of Bath '))
y = float(input('y of Bath '))
carpet_area = carpet_area + x*y
#Balcony
x = float(input('x of Balcony '))
y = float(input('y of Balcony '))
carpet_area = carpet_area + x*y
print('area of 1 BHK flat', carpet_area)
```

```
# compute "carpet area" of 1BHK flat

def room_area(room):
    x = float(input('x of ' + room))
    y = float(input('y of ' + room))
    return(x*y)

carpet_area = 0
#Drawing+dining
carpet_area = carpet_area + room_area('DD_room')
#Bedroom
carpet_area = carpet_area + room_area('Bedroom')
#Kitchen
carpet_area = carpet_area + room_area('Kitchen')
#Bath
carpet_area = carpet_area + room_area('Bath')
#Balcony
carpet_area = carpet_area + room_area('Balcony')

print('carpet area of 1 BHK flat', carpet_are
```

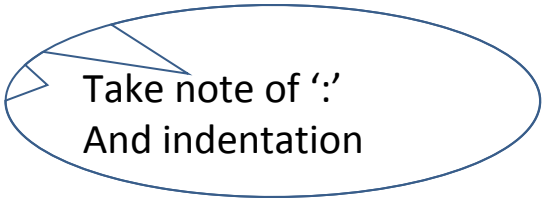
Functions, or sub-programs

- A program with functions will look something like this:

```
#Python program with a function, func1
def func1(formal parameters):
body_of_func1
```

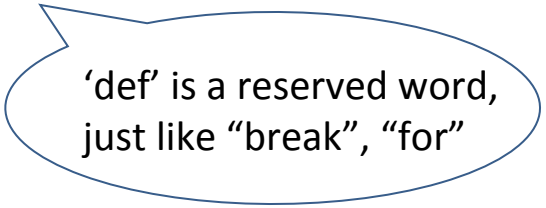
This is how one "calls" a function

```
S1
S2
S3
func1(actual parameters) #this is how you call or invoke func1
S4
S5
```



OR, when the function returns a value

```
S1
S2
S3
print(func1(actual parameters) #do something with returned value
S4
S5
```

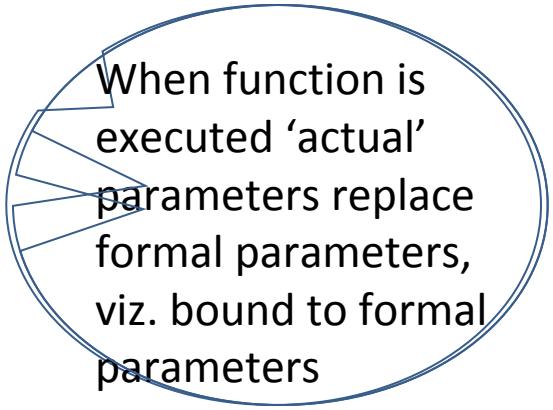


Functions, or sub-programs

- Example function, and function call:

```
#function returns the larger of x and y
def maxVal(x,y):
 if x > y:
        return x
    else:
        return y

   Some_statement_1
print(maxVal(45, 66))
Some_statement_2
print(maxVal(76, 19))
Some_statement_3
```

```
Output:
 66
 76
```



Functions, or sub-programs

- Example function, and function call:

```
#function returns the larger of x and y
def maxVal(x,y):
 if x > y:
        return x
    else:
        return y

   Some_statement_1
print(maxVal(45, 66))
Some_statement_2
print(maxVal(76, 19))
Some_statement_3
```

```
Output:
    66
    76
```

When function is executed 'actual' parameters replace formal parameters, viz. bound to formal parameters

Functions, or sub-programs

- Parameter binding
  - Positional binding: the first formal parameter is bound to the first actual parameter, the second one to the ...
  - Binding using name: the name of the formal parameter is used to do the binding

```
#Python Function Definition
def printName(first, last, reverse):
    if reverse:
        print(last + ', ' + first)
    else:
        print(first, last)


#Python Function Call
#All the following calls are equivalent

oprintName('Jack', 'Sparrow', False)
oprintName('Jack', 'Sparrow', reverse = False)
oprintName('Jack', last = 'Sparrow', reverse = False)
oprintName(last = 'Sparrow', first = 'Jack', reverse = False)
```

Functions, or sub-programs

- Parameter binding
  - Positional binding: the first formal parameter is bound to the first actual parameter, the second one to the ...
  - Binding using name: the name of the formal parameter is used to do the binding

```
#Python Function Definition
def printName(first, last, reverse):
    if reverse:
        print(last + ', ' + first)
    else:
        print(first, last)
```

Output if reverse = False:

```
Jack Sparrow
```

Output if reverse = True:

```
Sparrow, Jack
```

```
#Python Function Call
#All the following calls are equivalent

oprintName('Jack', 'Sparrow', False)
oprintName('Jack', 'Sparrow', reverse = False)
oprintName('Jack', last = 'Sparrow', reverse = True)
oprintName(last = 'Sparrow', first = 'Jack', reverse = True)
```

Functions, or sub-programs

- Parameter binding
  - Positional binding: the first formal parameter i… al parameter, the second one to the …
  - Binding using name: the name of t… do the binding

Either method is fine. But, it is best that we maintain the order of the actual parameters to avoid mistakes

```
#Python Function Definition
def printName(first, last, revers
    if reverse:
        print(last + ', ' + first)
    else:
        print(first, last)
```

Output if reverse = False:

```
Jack Sparrow
```

Output if reverse = True:

```
Sparrow, Jack
```

```
#Python Function Call
#All the following calls are equivalent

oprintName('Jack', 'Sparrow', False)
oprintName('Jack', 'Sparrow', reverse = False)
oprintName('Jack', last = 'Sparrow', reverse = True)
oprintName(last = 'Sparrow', first = 'Jack', reverse = True)
```

Functions, or sub-programs

- Parameter binding
  - An actual parameter value can be assigned to the formal parameters while defining function
    - This becomes the default value.
  - This allows the function to be called with fewer actual parameters

```
#Python Function Definition
def printName(first, last, reverse = False):
    if reverse:
        print(last + ', ' + first)
    else:
        print(first, last)
```

Defining the default value for parameter

- **printName** can be invoked or called as follows:

```
printName('Jack', 'Sparrow')
printName('Jack', 'Sparrow', True)
printName('Jack', 'Sparrow', reverse = True)
```

Default value applies

Default value is over-ridden

Functions, or sub-programs

- Another example application: Compute the grade of students in my class:
  - Requires 3 functions
    1. compute_total to compute total marks, out of 100
    2. minT to compute minimum of T1, T2, T3, T4
    3. compute_grade to compute the grade

```
# 1st function compute_total to compute total marks, out of 100
def compute_total(L1, L2, L3, L4, E1, E2):
    min_labs = minT(L1, L2, L3, L4)
    total_labs = (L1 + L2 + L3 + L4 - min_labs) /60 * 40
    total_exams = (E1 + E2) / 180 * 60
    return(total_labs + total_exams)

# 2nd function minT to compute minimum of T1, T2, T3, T4
def minT(T1, T2, T3, T4):
    minT = T1
    if T2 < minT:
        minT = L2
    if T3 < minT:
        minT = L3
    if T4 < minT:
        minT = L4
    return minT
```

Functions, or sub-programs

```
# 1st function compute_total to compute total marks, out of 100
def compute_total(L1, L2, L3, L4, E1, E2):
    ETC. ETC.

# 2nd function minT to compute minimum of T1, T2, T3, T4
def minT(T1, T2, T3, T4):
    ETC. ETC.

# 3rd function compute_grade to  compute the grade
def compute_grade(marks):
    if marks >= 80:
        return('A')
    if marks >= 65:
        return('B')
    if marks >= 50:
        return('C')
    if marks >= 35:
        return('D')
    if marks < 35:
        return('F')

# main block
roll_no = input('roll no.: ')
Lab1, Lab2, Lab3, Lab4 = input('Enter marks in Lab1, Lab2, Lab3, Lab4 ').split()
Exam1, Exam2 = input('Enter marks in Exam1, Exam2 ').split()
```

Functions, or sub-programs

```
# 1st function compute_total to compute total marks, out of 100
def compute_total(L1, L2, L3, L4, E1, E2):
    ETC. ETC.

# 2nd function minT to compute minimum of T1, T2, T3, T4
def minT(T1, T2, T3, T4):
    ETC. ETC.

# 3rd function compute_grade to  compute the grade
def compute_grade(marks):
    ETC. ETC.

# main block
roll_no = input('roll no.: ')
Lab1, Lab2, Lab3, Lab4 = input('Enter marks in Lab1, Lab2, Lab3, Lab4 ').split()
Exam1, Exam2 = input('Enter marks in Exam1, Exam2 ').split()
total = compute_total(int(Lab1), int(Lab2), int(Lab3), int(Lab4), int(Exam1), int(Exam2))
print('Total: ' , total, 'Grade: ', compute_grade(total))
```

Output:

```
roll no.: 1234
Enter marks in Lab1, Lab2, Lab3, Lab4 13 15 9 10
Enter marks in Exam1, Exam2 45 90
B
```

Functions, or sub-programs

```
# 1st function compute_total to compute total marks, out of 100
def compute_total(L1, L2, L3, L4, E1, E2):
    ETC. ETC.

# 2nd function minT to compute minimum of T1, T2, T3, T4
def minT(T1, T2, T3, T4):
    ETC. ETC.

# 3rd function compute_grade to  compute the grade
def compute_grade(marks):
    ETC. ETC.
```

https://tinyurl.com/3edxvsy2

```
# main block
roll_no = input('roll no.: ')
Lab1, Lab2, Lab3, Lab4 = input('Enter marks in Lab1, Lab2, Lab3, Lab4 ').split()
Exam1, Exam2 = input('Enter marks in Exam1, Exam2 ').split()
total = compute_total(int(Lab1), int(Lab2), int(Lab3), int(Lab4), int(Exam1), int(Exam2))
print('Total: ' , total, 'Grade: ', compute_grade(total))
```

Output:

```
roll no.: 1234
Enter marks in Lab1, Lab2, Lab3, Lab4 13 15 9 10
Enter marks in Exam1, Exam2 45 90
B
```

Functions, or sub-programs

Yet another example application of functions: Determine whether a given N is prime or not

MAIL IIITD HDFC bank Bank PayTM SBI Direct

```
1  # determine whether given N is prime or not
2  def sqrt(K):
3      tolerance, lower, upper = 0.001, 0.0, K
4      uncertainty = upper-lower;
5      while uncertainty > tolerance:
6          middle = (lower + upper)/2
7          if middle**2 < N:
8              lower = middle
9          else:
10             upper = middle
11         print(lower, upper);
12         uncertainty = upper-lower
13     return((lower+upper)/2)
14 # the main program
15 N = int(input('N: '))
16 if N > 3:
17     max = int(sqrt(N))+1
18     print('Checking for divisibility from 2 to ', max)
19     composite = False
20     for k in range(2, max+1):
21         if N%k == 0:
22             composite = True
23             print('N = ', N, 'is a composite no.')
24             break
25     if composite == False:
26         print('N = ', N, 'is a prime no.')
27 else:
28     print('N = ', N, 'is a prime no.')
```

https://tinyurl.com/y7d5yy

Functions, or sub-programs

Yet another example application of functions:
Compute the largest prime number <= BIG

```
# determine largest prime no equal to or less than BIG
def sqrt(K):
    etc. etc.
def IS_Prime(N):                    # assumed N > 3
    max = int(sqrt(N))+1
    print('Checking for divisibility from 2 to ', max)
    composite = False
    for k in range(2, max+1):
        if N%k == 0:
            composite = True
            print('N = ', N, 'is a composite no.')
            return(False)
    if composite == False:
        print('N = ', N, 'is a prime no.')
        return(True)
# The main program
BIG = int(input('BIG: '))
print(BIG)
for M in range(BIG, 6, -1):
    if IS_Prime(M):
        print('Largest prime <= ', BIG, M)
        break
```

https://tinyurl.com/54jkdxk8

Functions, or sub-programs

```
# determine the average distance between two prime numbers
# that are in range SMALL to BIG (both inclusive)
def sqrt(K):
    etc. etc.
def IS_Prime(N):                    # assumed N > 3
    etc. etc.
def next_Prime(K):                  #find the next Prime number > K
    j = K+1
    while IS_Prime(j) == False:
        j = j+1
    return(j)
# The main program
small, big = input('small and big: ')split()
print('small:', small, 'big: ', big)
sum_total_distance, no_primes = 0, 0
first_Prime = next_Prime(small-1)
last_Prime = first_Prime
no_Primes = 1
while last_Prime < big:
    p = next_Prime(last_Prime)
    if p <= big:
        sum_total_distance = sum_total_distance + (p - last_Prime)
        no_primes = no_primes + 1
    last_prime = p
average_dist = sum_total_distance/(no_Primes -1)
print('average distance: ', average_dist)
```

NOT TESTED: encourage you to test this

Scope of variables

- Local variables
  - Variables declared within a function (or within a block) are available only within that function or block
- Global variables
  - Variables declared outside of a function are accessible as "global" variables
    - BUT only if you do NOT give same names to the local & global variable

```
def f(x):    #x is formal parameter, and therefore local to f(.)
    y = 1    #y defined within f(.), and therefore local to f(.)
    x = x + y
    print('f(.) x =', x)
    return(x)

x = 3 #x is local to "main"
y = 2 #y is local to "main"
z = f(x)   #value of x is actual parameter. z is also local to "main"
print('Main z =', z)
print('Main x =', x)
print('Main y =', y)
```

Also referred to as "main" program

Scope of variables

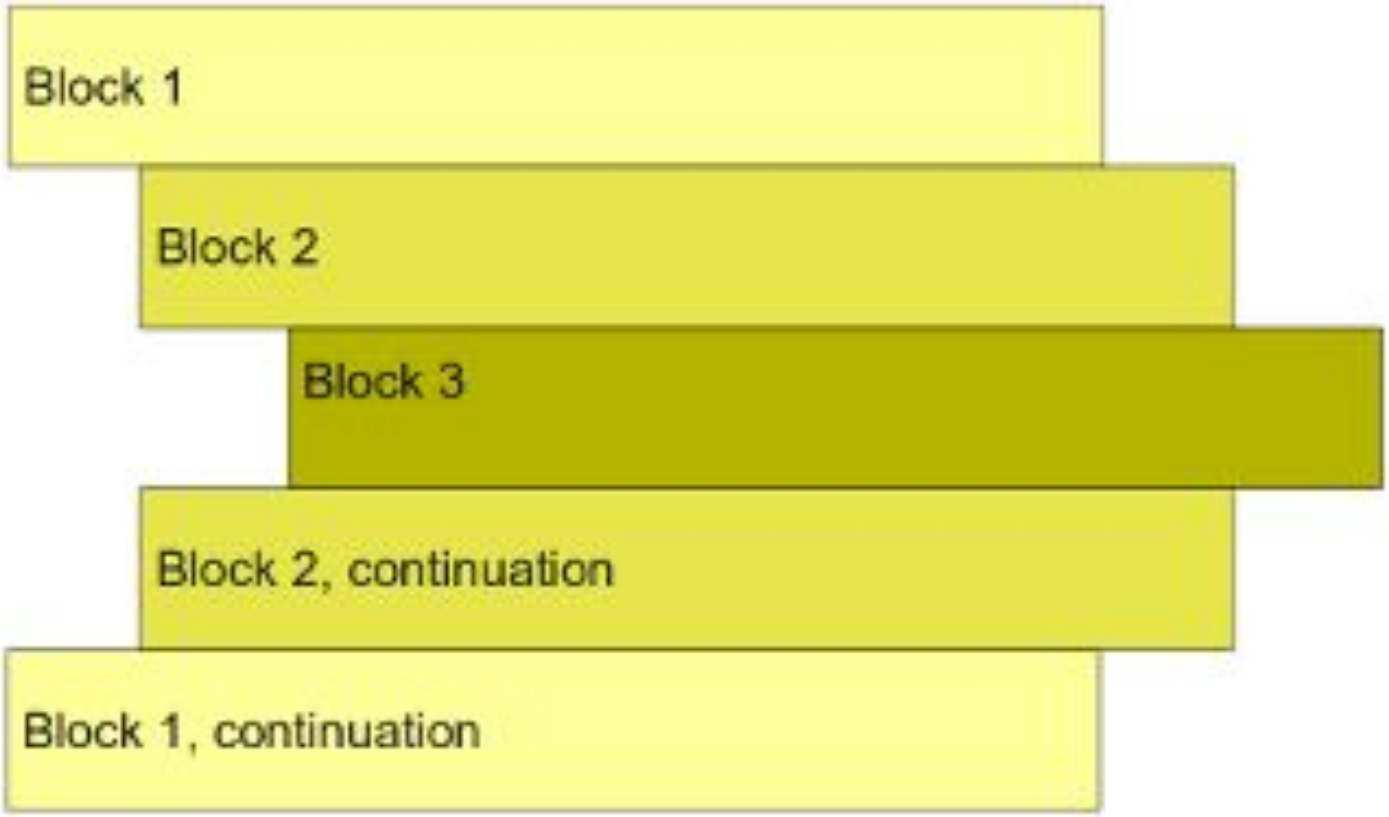
```
def f(x):    #x is formal parameter, and therefore local to f(.)
    y = 1    #y defined within f(.), and therefore local to f(.)
    x = x + y
    print('f(.) x =', x)
    return(x)

x = 3 #x is local to "main"
y = 2 #y is local to "main"
z = f(x)   #value of x is actual parameter. z is also local to "main"
print('Main z =', z)
print('Main x =', x)
print('Main y =', y)
```

https://tinyurl.com/4efkb3f3

```
When run, this code prints:
f(.) x = 4
Main z = 4
Main x = 3
Main y = 2
```

A digression to code blocks to understand scope

Code blocks


Block 1
Block 2
Block 3
Block 2, continuation
Block 1, continuation

Scope of variables

- Local variables
  - Variables declared within a function (or within a block) are only available within that block or function
- Global variables
  - Variables declared outside of a function are accessible as "global" variables
    - BUT only if you do NOT give same names to the local & global variable

f(.)

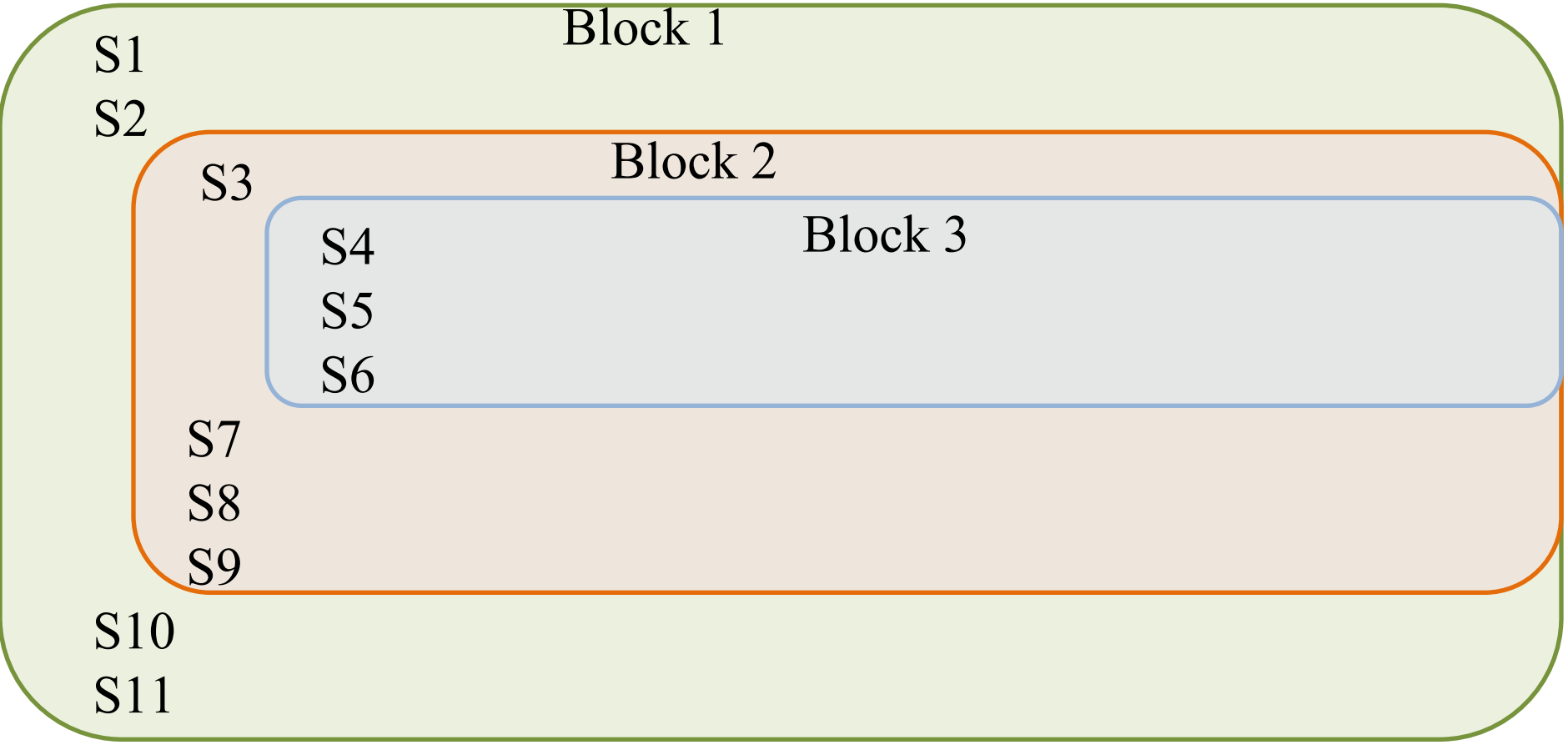
```
def f(x):    #x is formal parameter, and therefore local to f(.)
    y = 1    #y defined within f(.), and therefore local to f(.)
    x = x + y
    print('x =', x)
    return(x)
```

main

```
x = 3 #x is local to "main"
y = 2 #y is local to "main"
z = f(x)   #value of x is actual parameter. z is also local to "main"
print('z =', z)
print('x =', x)
print('y =', y)
```

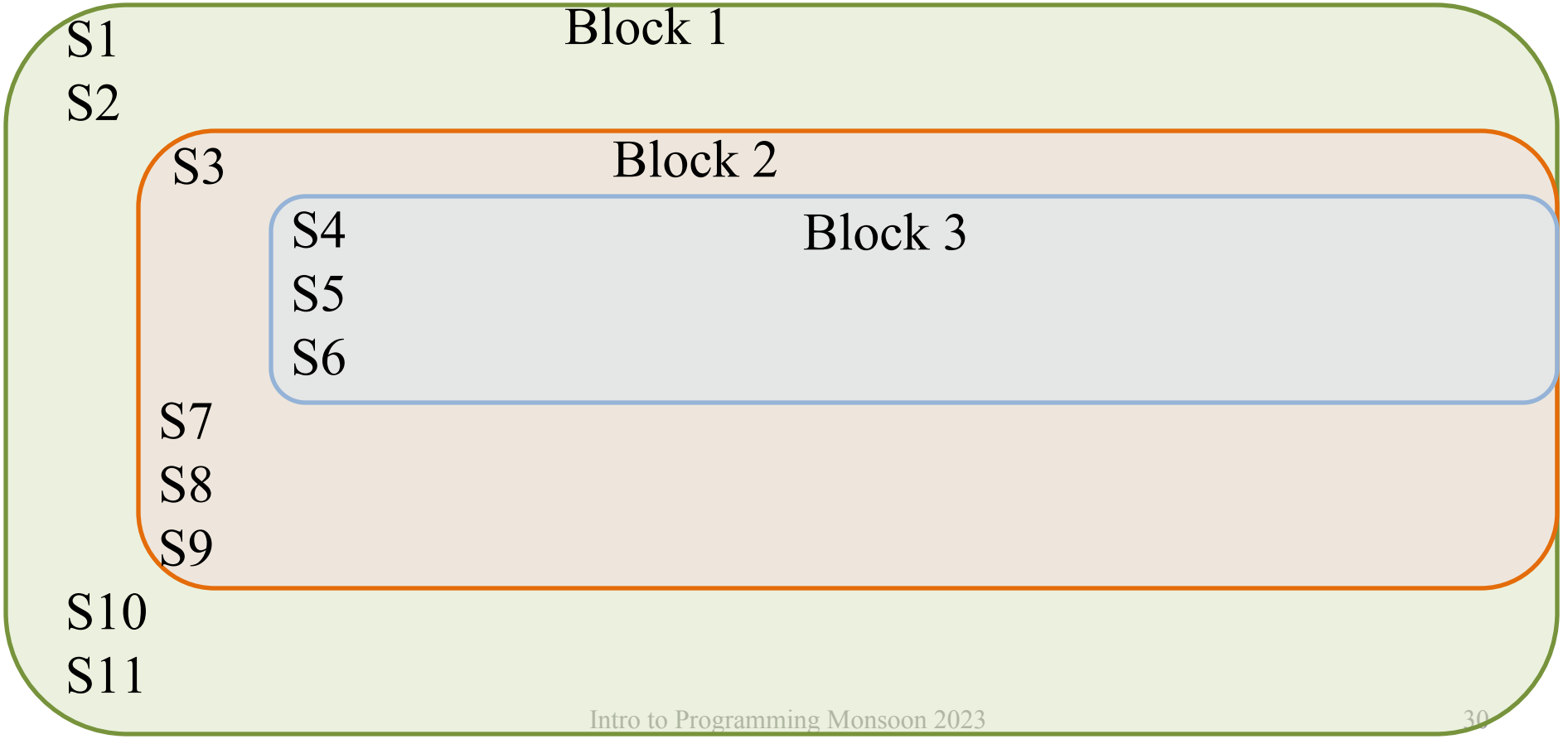
	Local variables/objects	Global variables/objects
Main	x, y, z, f(.)	---
f(.)	x, y	z

Code blocks



Code blocks

- Statements that can be grouped together are called “block” or a “code block”
- Statements inside a block can be treated as one logical statement
- Blocks can be nested inside each other
- Blocks are clearly delimited using indentation in Python
- Other programming languages like C, C++, Java use { }, []



Code blocks

- Statements that can be grouped together are called “block” or a “code block”
- Statements inside a block can be treated as one logical statement
- Blocks can be nested inside each other
- Blocks are clearly delimited using indentation in Python
- Other programming languages like C, C++, Java use { }, []

```
1   def blockDemo(x):
2       n = int(x) + 1 #x, n are local to blockDemo
2       for a in range(1, n): #a is local to “for” block, n is global
3           for b in range(a,n): #b is local to this block, a, n global
4               c_square = a**2 + b**2 #c_square is local
4               c = int(sqrt(c_square)) #c is local
4               if ((c_square - c**2) == 0):
                    print(a, b, c)
3           y = a + int(x) #y,a are local, x is global, b not accessible
2       n = n*2 #a is not accessible
1   x = ’19’ #x is actual parameter, local to “main”
1   blockDemo(x)
```

blockDemo

for a …

for b …

main

Code blocks

```
def blockDemo(x):
    n = int(x) + 1 #x, n are local to blockDemo
    for a in range(1, n): #a is local to "for" block, n is global
        for b in range(a,n): #b is local to this block, a, n global
            c_square = a**2 + b**2 #c_square is local
            c = int(sqrt(c_square)) #c is local
            if ((c_square - c**2) == 0):
                print(a, b, c)
        y = a + int(x) #y,a are local, x is global,b not accessible
    n = n*2 #a is not accessible

x = '19' #x is actual parameter, local to "main"
blockDemo(x)
```

blockDemo

for a …

for b …

main

Code blocks

	Local variables/objects	Global variables/objects
Main	x, blockDemo	
blockDemo	x, n	
for a …	a, y	n, x
for b …	b, c, c_square	a, n

main

blockDemo

for a …

for b …

```
1 def blockDemo(x):
2     n = int(x) + 1 #x, n are local to blockDemo
2     for a in range(1, n): #a is local to "for" block, n is global
3         for b in range(a,n): #b is local to this block, a, n global
4             c_square = a**2 + b**2 #c_square is local
4             c = int(sqrt(c_square)) #c is local
4             if ((c_square - c**2) == 0):
                  print(a, b, c)
3         y = a + int(x) #y, a are local, x is global
2     n = n*2 #a is not accessible
1 x = '19' #x is actual parameter, local to "main"
1 blockDemo(x)
```

Scope of variables

- A variable is considered to be a local variable if it is assigned value within the function
- Else, it is treated as a global variable in an outer block or function

```
def f():
    print(x)
def g():
    print(x)
    x = 1
x = 3
f()
x = 3
g()
```

x is global, since x appears on RHS

x is local, since x appears on LHS of a statement in g()

x is local to "main", since x appears on LHS

```
Output:
x = 3
Error: x is not assigned value
```

	Local variables/objects	Global variables/objects
Main	x, f(.), g(.)	
f(.)	x	
g(.)	x?	x?

Scope of variables

- A variable is considered to be a local variable if it is assigned value within the function
- Else, it is treated as a global variable in an outer block or function

```
def f():
    print(x)
def g():
    print(x)
    x = 1
x = 3
f()
x = 3
g()
```

x is global, since x appears on RHS

x is local, since x appears on LHS of a statement in g()

x is local to "main", since x appears on LHS

```
Output:
x = 3
Error: x is not assigned value
```

	Local variables/objects	Global variables/objects
Main	x, f(.), g(.)	
f(.)	x	
g(.)	x?	x?

Scope of variables

- Local variables
  - Variables declared within a block or within a function are only available within that block or function
- Global variables
  - Variables that are declared outside of a function -- avoid giving same names to local & global variables

```
Program:
def f(x):
    def g():
        x = 2000
        print('g(.) x =', x)
    def h():
        z = x
        print('h(.) z =', z)
    x = x + 1
    print('f(.) x =', x)
    h()
    g()
    print('f(.) x =', x)
    return(x)
x = 3
z = f(x)
print('main x =', x)
print('main z =', z)
```


Try experimenting the above code using http://pythontutor.com

Scope of variables

- Local variables
  - Variables declared within a block or within a function are only available within that block or function
- Global variables
  - Variables that are declared outside of a function -- avoid giving same names to local & global variables

x is local to f(.), since x is formal parameter

x is local, since x appears on LHS

x is global, since x appears on RHS, z is local to h(.)

x is local to "main", since x appears on LHS,
z is also local to "main"

```
Program:
def f(x):
    def g():
        x = 2000
        print('g(.) x =', x)
    def h():
        z = x
        print('h(.) z =', z)
    x = x + 1
    print('f(.) x =', x)
    h()
    g()
    print('f(.) x =', x)
    return(x)
x = 3
z = f(x)
print('main x =', x)
print('main z =', z)
```


Try experimenting the above code using http://pythontutor.com

Scope of variables

- Local variables
  - Variables declared within
- Global variables
- Variables that are declared outside of a function – avoid giving same names to local & global variables

	Local variables/objects	Global variables/objects
Main	x, z, f(.)	
f(.)	x, h(.), g(.)	z (of main), f(.)
g(.)	x	z (of main), h(.), g(.)
h(.)	z	x (of f(.)), h(.), g(.)

x is local to f(.), since x is formal parameter

x is local, since x appears on LHS

x is global, since x appears on RHS, z is local to h(.)

x is local to "main", since x appears on LHS,
z is also local to "main"

```
Program:
def f(x):
    def g():
        x = 2000
        print('g(.) x =', x)
    def h():
        z = x
        print('h(.) z =', z)
    x = x + 1
    print('f(.) x =', x)
    h()
    g()
    print('f(.) x =', x)
    return(x)
x = 3
z = f(x)
print('main x =', x)
print('main z =', z)
```


Try experimenting the above code using http://pythontutor.com

Scope of variables

- Local variables
  - Variables declared within a block or within a function are only available within that block or function
- Global variables
  - Variables that are declared outside of a function -- avoid giving same names to local & global variables

x is local to f(.), since x is formal parameter

x is local, since x appears on LHS

x is global, since x appears on RHS, z is local to h(.)

x is local to "main", since x appears on LHS,
z is also local to "main"

```
Program:
def f(x):
    def g():
        x = 2000
        print('g(.) x =', x)
    def h():
        z = x
        print('h(.) z =', z)
    x = x + 1
    print('f(.) x =', x)
    h()
    g()
    print('f(.) x =', x)
    return(x)
x = 3
z = f(x)
print('main x =', x)
print('main z =', z)
```

```
Sequence of actions/executions:
    x = 3
    f(.) is called
    Formal par x = 3
    Local x = x + 1 or 4
    x = 4 is printed
    h(.) is called
    Local z = x using global x (=4)
    z = 4 is printed, h ends
    g(.) is called
    Local x = 2000
    Local x = 2000 is printed
    g ends
    local x = 4 is printed
    Result x =4 is returned
    F(.) ends, Z = 4
    Main x = 3 printed
    z = 4 printed
```

Try experimenting the above code using http://pythontutor.com

Scope of variables

- Local variables
  - Variables declared within a block or within a f … hin that block or function
- Global variables
  - Variables that are declared outside of a functi … s to local & global variabl…

```
Program:
def f(x):
    def g():
        x = 2000
        print('g(.) x =', x)
    def h():
        z = x
        print('h(.) z =', z)
    x = x + 1
    print('f(.) x =', x)
    h()
    g()
    print('f(.) x =', x)
    return(x)
x = 3
z = f(x)
print('main x =', …
print('main z =', z)
```

x is local to f(.), since x is formal parameter

x is local, since x appears on LHS

x is global, since x appears on RHS, z is local to h(.)

x is local to "main", since x appears on LHS,
z is also local to "main"

```
Output:
    f(.) x = 4
    h(.) z = 4
    g(.) x = 2000
    f(.) x = 4
    main x = 3
    main z = 4
```

```
Sequence of actions/executions:
    x = 3
    f(.) is called
    Formal par x = 3
    Local x = x + 1 or 4
    x = 4 is printed
    h(.) is called
    Local z = x using global x (=4)
    z = 4 is printed, h ends
    g(.) is called
    Local x = 2000
    Local x = 2000 is printed
    g ends
    local x = 4 is printed
    Result x =4 is returned
    F(.) ends, Z = 4
    Main x = 3 printed
    z = 4 printed
```

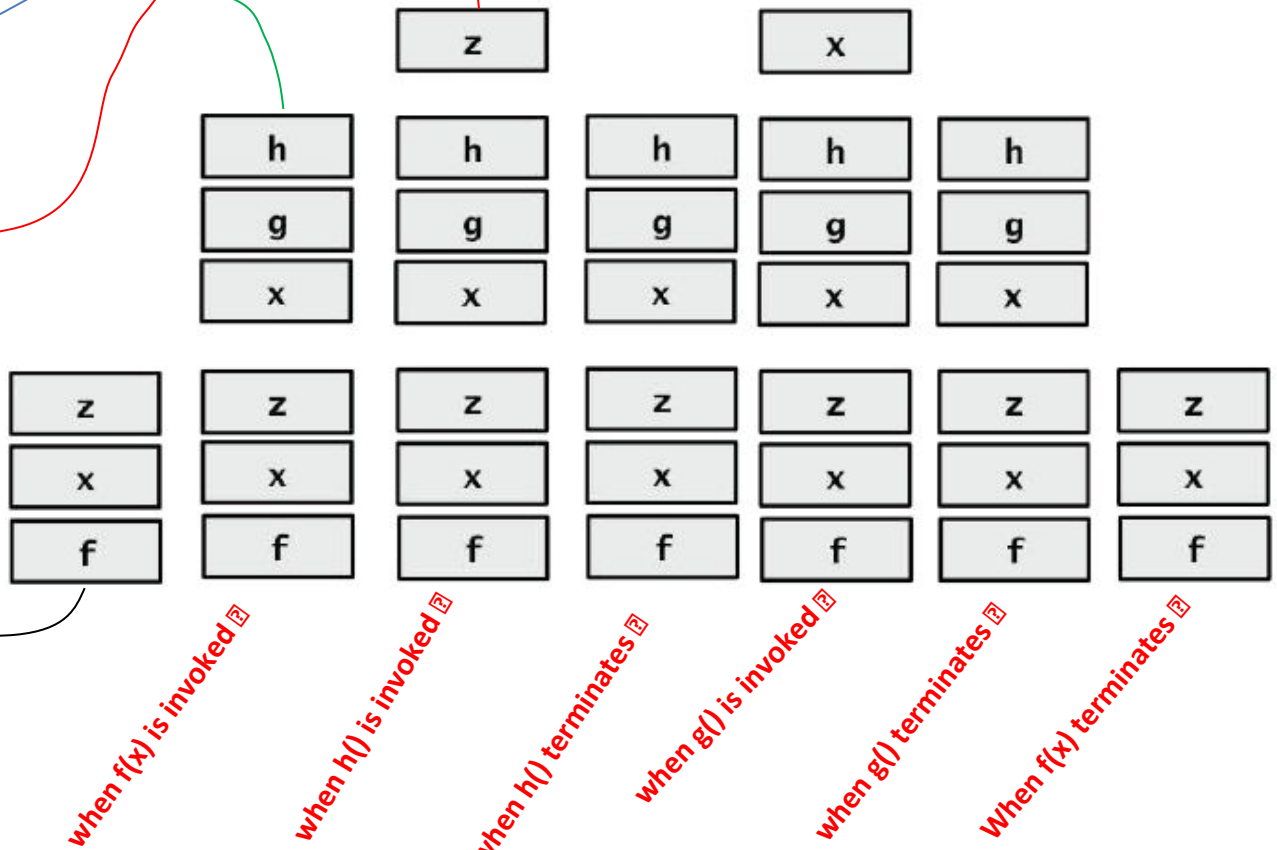

Try experimenting the above code using http://pythontutor.com

Stack frames

A frame lists names of objects known from inside the “main” program or a function
With nesting of main program & functions, the set of frames acts as a “stack” (“push” and “pop” operations

	Local variables/objects	Global variables/objects
Main	x, z, f(.)	
f(.)	x, h(.), g(.)	z (of main), f(.)
g(.)	x	z (of main), h(.), g(.)

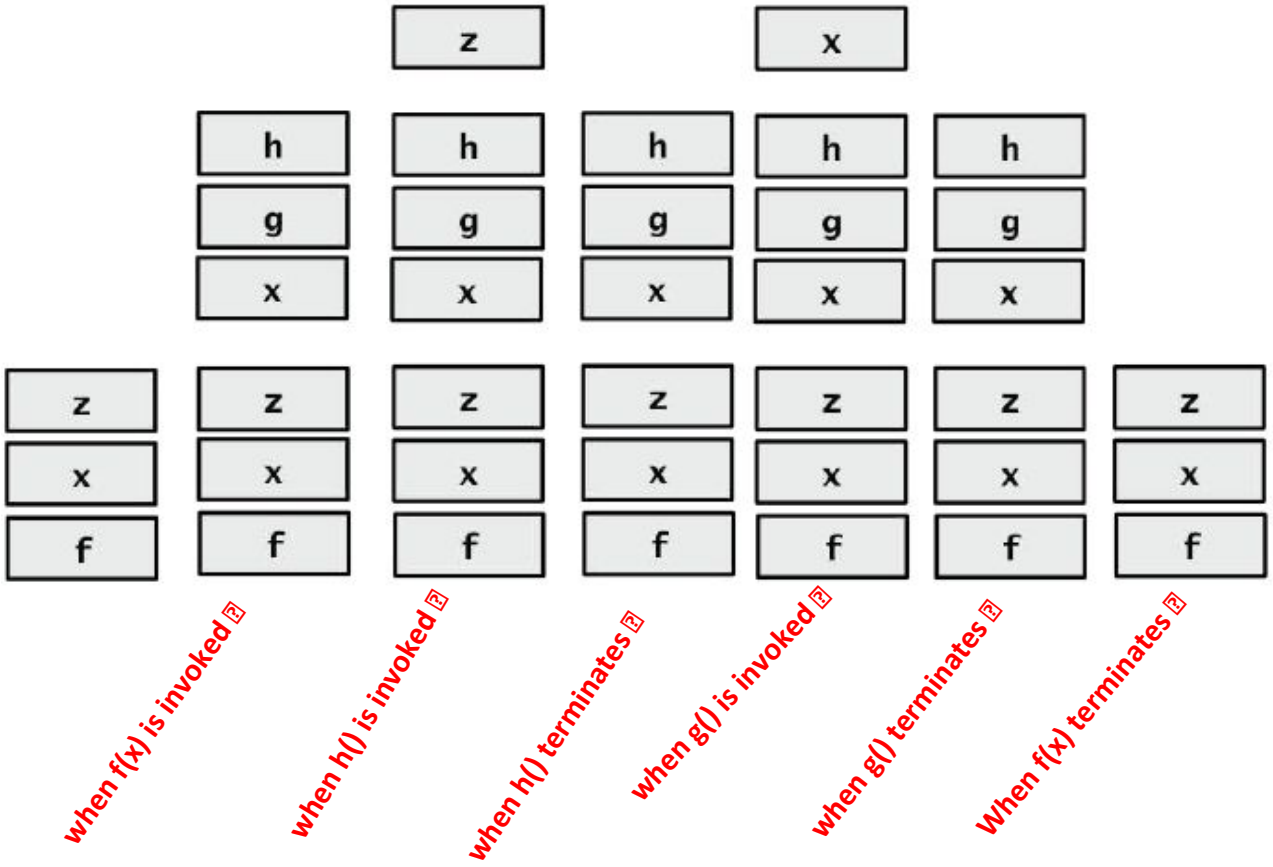
```
Program:
def f(x):
    def g():
        x = 2000
        print('x =', x)
    def h():
        z = x
        print('z =', z)
    x = x + 1
    print('x =', x)
    h()
    g()
    print('x =', x)
    return(x)
x = 3
z = f(x)
print('x =', x)
print('z =', z)
```



Stack frames

	Local variables/objects	Global variables/objects
Main	x, z, f(.)	
f(.)	x, h(.), g(.)	z (of main), f(.)
g(.)	x	z (of main), h(.), g(.)
h(.)	z	x (of f(.)), h(.), g(.)

See how PythonTutor executes the program, while keeping track of frames
https://tinyurl.com/yea82c6h



Functions

Functions can return a function name, not merely data objects

Program:

```
def f(x):
    def g():
        x = 2000
        print('x =', x)
    def h():
        z = x
        print('z =', z)
    x = x + 1
    print('x =', x)
    h()
    g()
    print('x =', x)
    return g
x = 3
z = f(x)
print('x =', x)
print('z =', z)
z()
```

Output:

```
x = 4
z = 4
x = 2000
x = 4
x = 3
z = <function g at 0x15b43b0>
x = 2000
>>>
```

Scope rules: a twist in the tail

- Static scoping:
  - Scope is determined by the block of code in which the name is defined
  - This is also known as lexical scoping
  - Python follows lexical scoping
  - Code on the right side prints 1 with lexical scoping

- Dynamic scoping
  - Scope is determined by most recent value assigned to the variable
  - This is also known as dynamic scoping
  - Python follows lexical scoping
  - Lisp supports dynamic scoping

```
def f(a):
    return x + a
def g():
    x = 2
    return f(0)
x = 1
print(g())
```

Q&A

- ?