Functions, Scoping and Abstraction

Chapter 4

Functions

- **Functions** allow us to generalize code that is specific
- What functions have we already seen?
  - `print()`
  - `abs()`
  - `input()`
  - `max()`
- **`def`** *`name of function`* `(`*`list of`* ***`formal parameters`***`)`
- Terminates when a **return** statement is executed
- Or no more statements to execute

Newton-Raphson as a function

```
Newton-Raphson for square root
def newton_raphson_square_root(k, episilon)
    #Find x such that x**2 - 24 is within epsilon of 0
    guess = k/2.0
    number_guesses = 1
    while abs(guess*guess - k) >= epsilon:
        guess = guess - (((guess**2) - k)/(2*guess))
        number_guesses += 1
    return guess
```

Calling a function

- `sq24 = newton_raphson_square_root(24, 0.01)`

- 24 and 0.01 are **actual parameters** or **arguments**
- Parameters create a **lambda abstraction**
  - Not “specific” objects but whatever is passed as arguments
- Arguments can be positional (most common) or keyword assigned
  - Keyword assignments can be in any order
  - Positional and keyword assignments can be mixed
  - Once a keyword is encountered the rest of the arguments must be keywords

Default parameters

- Formal parameter assigned a value in function definition
- If an actual parameter is not provided the default is used
  - `def my_function(a, b=3)`
  - `my_function(2, 3)`
  - `my_function(2)`
  - `my_function(a=2, b=3)`
  - `my_function(b=3, a=2)`
- Once a named parameter has been used, all remaing parameters must also be named
  - `my_function(b=3, 2)`

Variable parameters

- Some functions allow for an unknown number of parameters
  - `max(1,10)`
  - `max(1,2,3,4,5,6,7,8)`
- Specification
  - `def max(*nums):`

Functions as parameters

- Functions are objects
- Can be passed as arguments

Scoping

- A variable can exist in multiple scopes
- An expression will use the variable in the closest scope

Scoping Example

```
def f(x):
    y = 1
    x = x + y
    print 'x =', x
    return x
x = 3
y = 2
z = f(x)
Print('z =‘, z)
Print('x =‘, x)
Print('y =‘, y)
```

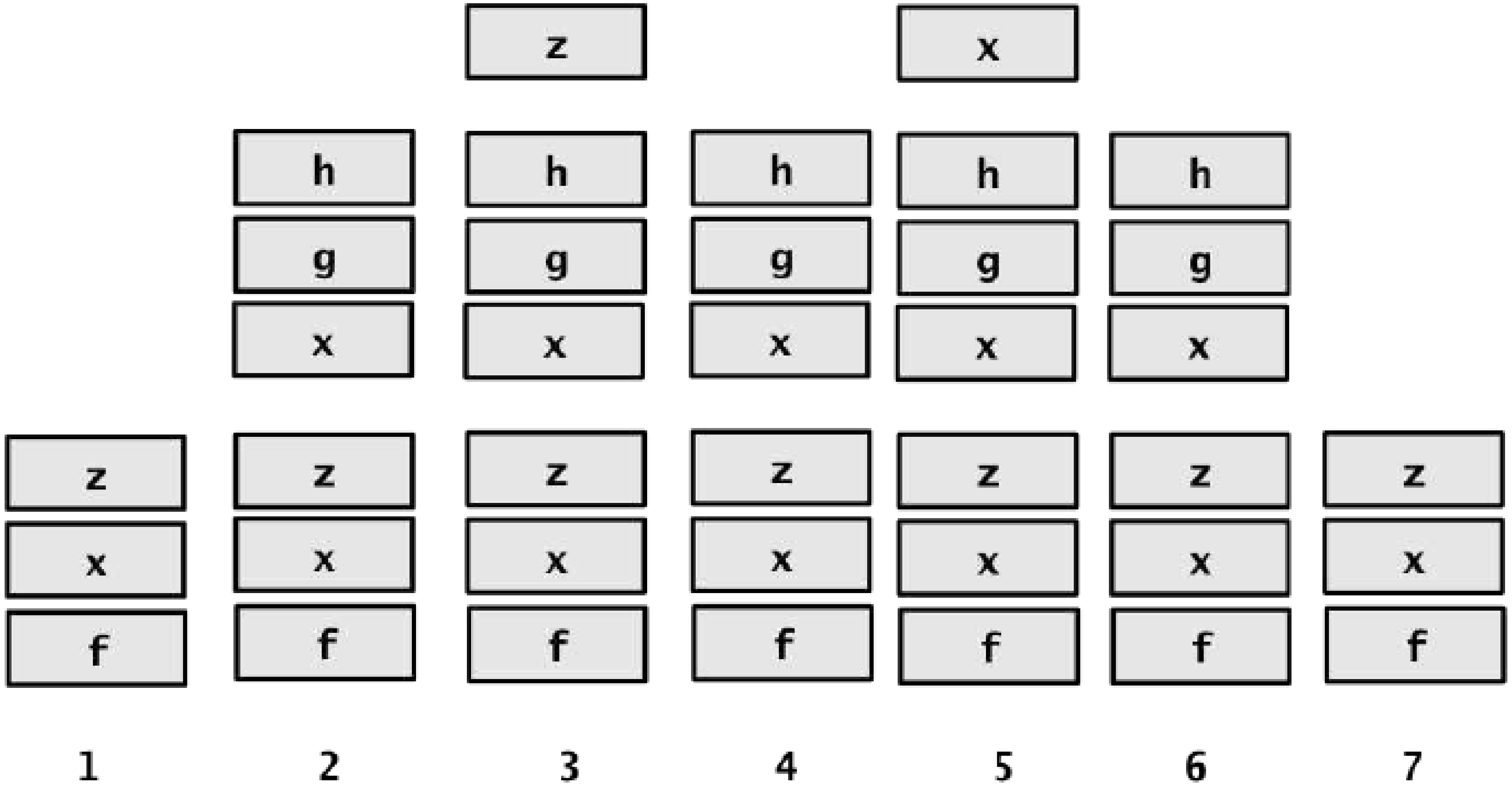
```
x = 4
z = 4
x = 3
y = 2
```

Another Scoping Example

```
def f(x):
    def g():
        x = 'abc’
        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()
```

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

Scoping & "the stack"



Specifications

- Documentation on a function
  - Assumptions
    - Expectations on users of the function
  - Guarantees
    - What the function will do – provided the assumptions are met
- Works with the Python help system
  - help(abs)

```
Help on built-in function abs in module __builtin__:
abs(...)
    abs(number) -> number
        Return the absolute value of the argument.
```

Specification

```
def find_root(x, power, epsilon):
    """Assumes x and epsilon int or float, power an int,
           epsilon > 0 & power >= 1
       Returns float y such that y**power is within epsilon
           of x.
       If such a float does not exist, it returns None"""
    if x < 0 and power%2 == 0:
        return None
    low = min(-1.0, x)
    high = max(1.0, x)
    ans = (high + low)/2.0
    while abs(ans**power - x) >= epsilon:
       if ans**power < x:
    low = ans
       else:
    high = ans
    ans = (high + low)/2.0
    return ans
```

Benefits of functions

- Testing – more in two weeks
- Decomposition
  - Breaking a problem down into smaller problems
  - Easier to develop, understand and maintain
  - (hint: in Notebook put functions in their own cell)
- Abstraction
  - Functions can be more general purpose that straight line code
  - Logic in a function can be changed without affecting the rest of a program
- Reuse

Functions versus Methods

- Methods are very similar to functions
- Methods are properties of ‘objects’
- Use ‘dot notation’
- More when we talk about classes