# Functions

- So far, have seen numbers, assignments, input/ output, comparisons, looping constructs
  - Sufficient to be **Turing Complete**
- But code lacks abstraction
  - Have to reload file every time want to use
  - Can't use same variable names in other pieces of code
  - Can quickly get cumbersome to read and maintain
- Functions give us abstraction – allow us to capture computation and treat as if primitive

# Capturing computation as a function

- Idea is to encapsulate this computation within a scope such that can treat as primitive
  - Use by simply calling name, and providing input
  - Internal details hidden from users
- Syntax

```
def <function name> (<formal parameters>):
    <function body>
```

- def is a keyword
- Name is any legal Python name
- Within parenthesis are zero or more formal parameters – each is a variable name to be used inside function body

# A simple example

- Suppose we want to z to be the maximum of two numbers, x and y
- A simple script would be

```
if x > y:
    z = x
else:
    z = y
```

# A simple example

```
def max(x, y):
    if x > y:
        return x
    else:
        return y
```

- We can then invoke by

```
z = max(3, 4)
```

- When we call or invoke max(3, 4), x is bound to 3, y is bound to 4, and then body expression(s) are evaluated

# Function returns

- Body can consist of any number of legal Python expressions
- Expressions are evaluated until
  - Run out of expressions, in which case special value None is returned
  - Or until special keyword return is reached, in which case subsequent expression is evaluated and that value is returned as value of function call

# Environments to understand bindings

- Environments are formalism for tracking bindings of variables and values
- Assignments pair name and value in environment
- Asking for value of name just looks up in current environment
- Python shell is default (or global) environment
- Definitions pair function name with details of function

# Summary of function call

1. Expressions for each parameter are evaluated, bound to formal parameter names of function
2. Control transfers to first expression in body of function
3. Body expressions executed until return keyword reached (returning value of next expression) or run out of expressions (returning None)
4. Invocation is bound to the returned value
5. Control transfers to next piece of code

```
x = 5
p = 3

result = 1

for turn in range(p):
    print('iteration: ' + str(turn) + 'current result: ' +
  str(result))
    result = result * x
```

| | |
|---|---|
| x | 5 |
| p | 3 |
| result | 1 |

```
x = 5
p = 3

result = 1

for turn in range(p):
    print('iteration: ' + str(turn) + 'current result: ' +
   str(result))
    result = result * x
```

| x | 5 |
|---|---|
| p | 3 |
| result | 125 |

# When we call a function

- Want to evaluate `<expr0>(<expr1>, …, <exprn>)`
- First evaluate `<expr0>`, which looks up procedure object in environment
- Then evaluate each of the other `<expri>` to get values of parameters
- Bind parameter names in procedure object to values of arguments in a new frame, which has as a parent the environment in which procedure was defined
- Evaluate body of procedure relative to this new frame

# Back to functions

```
x = 5
y = 3

def max(x, y):
    if x > y:
        return x
    else:
        return y
```

| x | 5 |
|---|---|
| y | 3 |
| max | |

```
Procedure1
  (x, y)
   if x > y:
        return x
    else:
        return y
```

# When we call the function

```
x = 5
y = 3

def max(x, y):
    if x > y:
        return x
    else:
        return y

z = max(3, 4)
```

E1

| x | 5 |
|---|---|
| y | 3 |
| Max | |

```
Procedure1
  (x, y)
   if x > y:
        return x
    else:
        return y
```

E2

| x | 3 |
|---|---|
| y | 4 |

## Another simple example

- Suppose we want to compute powers of a number by successive multiplication
- Idea would be to keep track of number of multiplications, plus intermediate product
- Stop when have multiplied number x by itself p times, and return final product
- Here is simple code

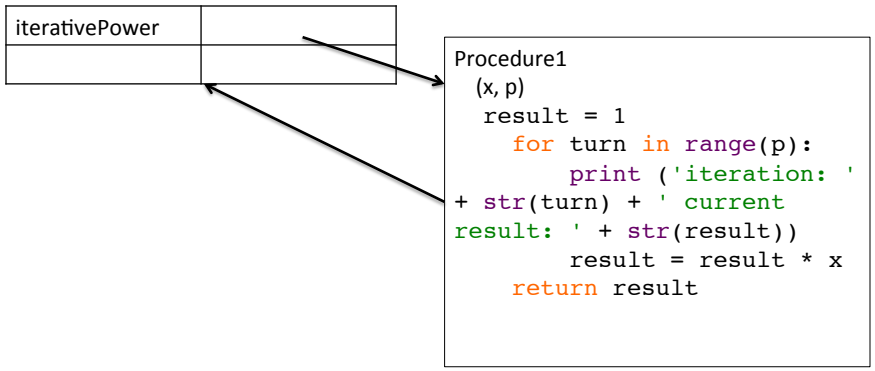
## Let's define our procedure

```
def iterativePower(x, p):
    result = 1
    for turn in range(p):
        print ('iteration: ' +
 str(turn) + ' current result: '
 + str(result))
        result = result * x
    return result
```

## Computing powers

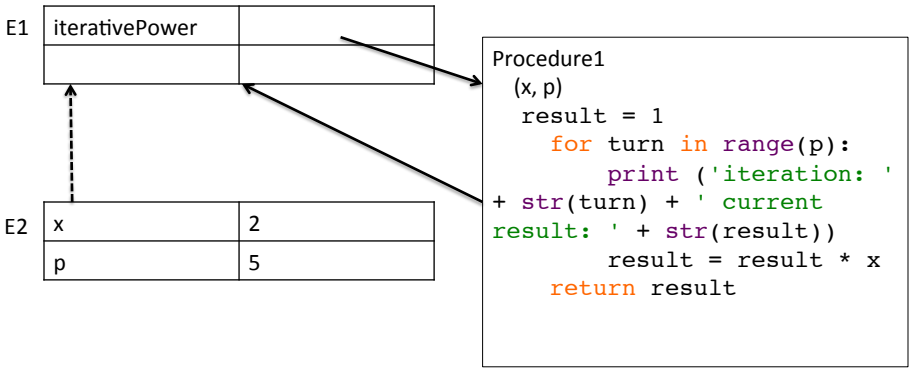
```
x = float(raw_nput('Enter a number: ')
p = int(raw_input('Enter an integer power: '))

result = 1

for turn in range(p):
    print('iteration: ' + str(turn) + \
          'current result: ' + str(result))
    result = result * x
```

## And this creates



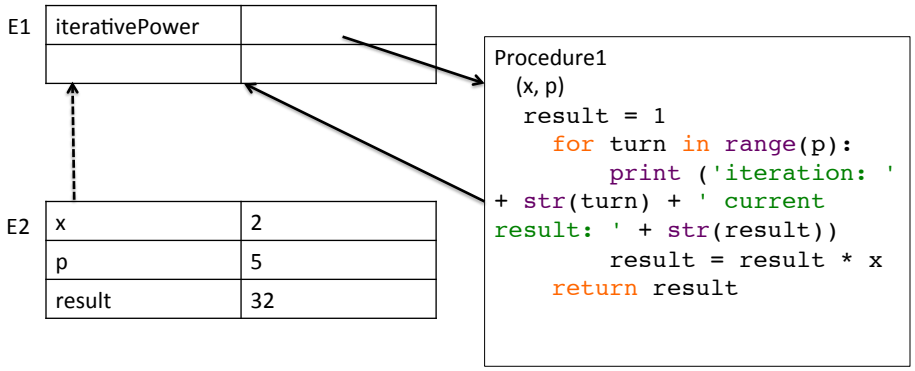
# Example

- Call iterativePower(2, 5)



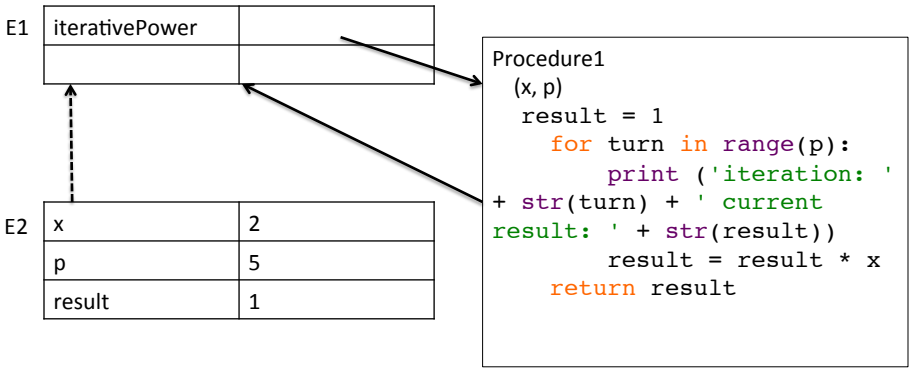

# Example

- and for loop rebinds local variable until exit, when return statement returns value of result




# Example

- Evaluating body of procedure initially causes…




# Scoping is local

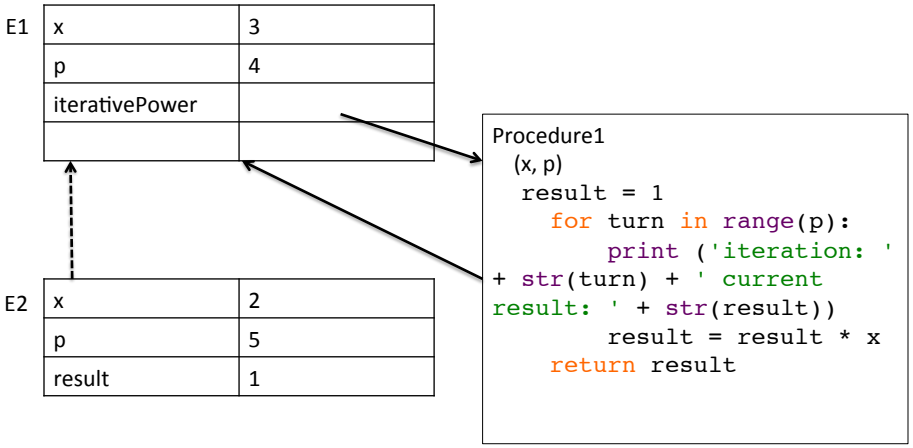
- Imagine we had evaluated

```
x = 3
p = 4
print(iterativePower(2, 5))
```

- Then our local environment would have separate bindings for x and p, which would not be visible to the environment of the function call

# Example

- Evaluating body of procedure initially causes…




# Another example

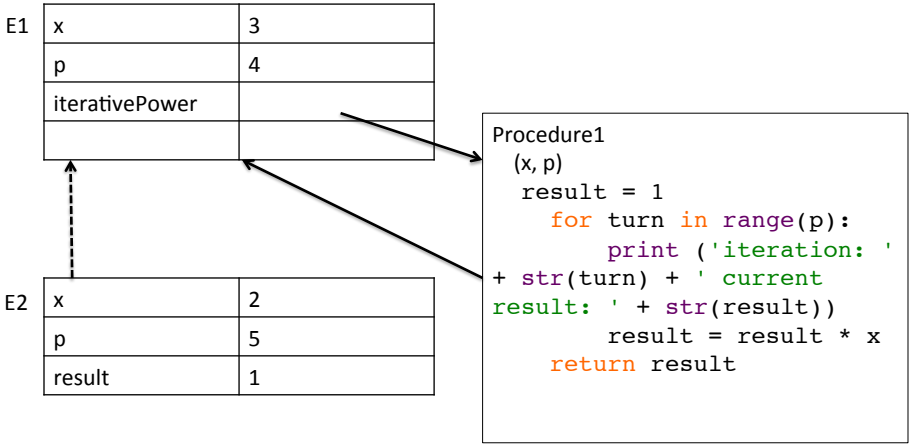
```
def f(x):
    y = 1
    x = x + y
    print('x = ' + str(x))
    return x

x = 3
y = 2
z = f(x)
print('z = ' + str(z))
print('x = ' + str(x))
print('y = ' + str(y))
```

- Causes the following to appear in the Python shell

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

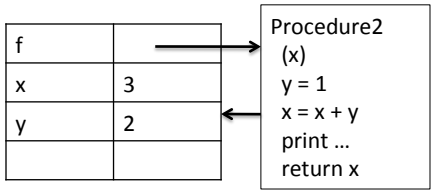
# Example

- But now evaluation of body only sees bindings in E2



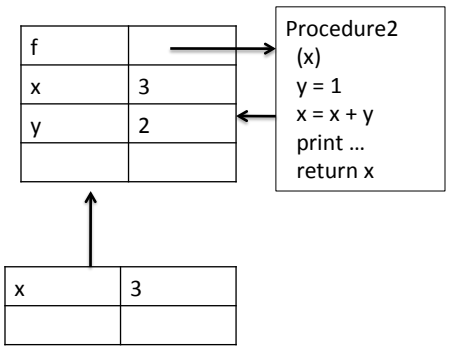

# Let's see why

```
def f(x):
    y = 1
    x = x + y
    print('x = ' = str(x))
    return x

x = 3
y = 2
z = f(x)
print('z = ' + str(z))
print('x = ' + str(x))
print('y = ' + str(y))
```



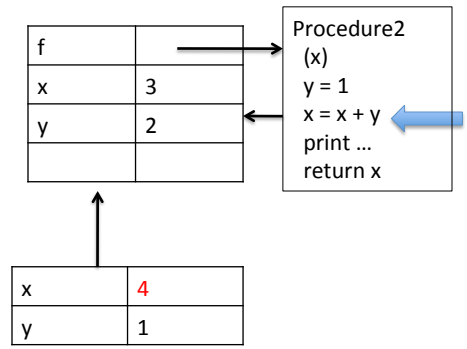
# Let's see why

```
def f(x):
    y = 1
    x = x + y
    print('x = ' = str(x))
    return x

x = 3
y = 2
z = f(x)
print('z = ' + str(z))
print('x = ' + str(x))
print('y = ' + str(y))
```



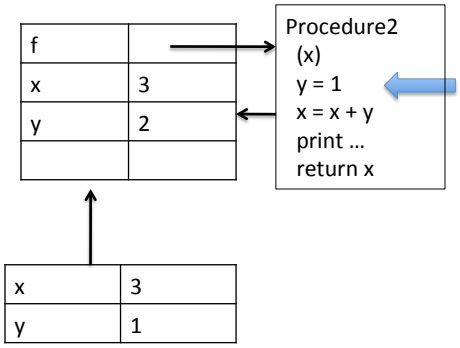

# Let's see why

```
def f(x):
    y = 1
    x = x + y
    print('x = ' = str(x))
    return x

x = 3
y = 2
z = f(x)
print('z = ' + str(z))
print('x = ' + str(x))
print('y = ' + str(y))
```



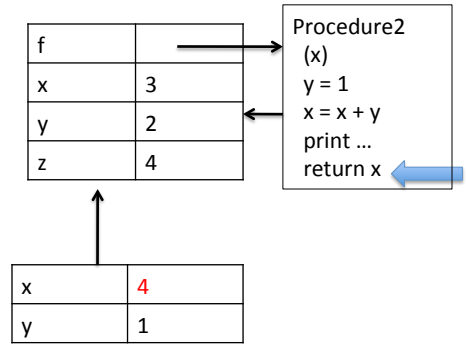

# Let's see why

```
def f(x):
    y = 1
    x = x + y
    print('x = ' = str(x))
    return x

x = 3
y = 2
z = f(x)
print('z = ' + str(z))
print('x = ' + str(x))
print('y = ' + str(y))
```



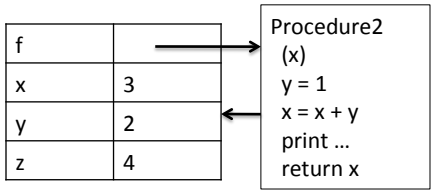

# Let's see why

```
def f(x):
    y = 1
    x = x + y
    print('x = ' = str(x))
    return x

x = 3
y = 2
z = f(x)
print('z = ' + str(z))
print('x = ' + str(x))
print('y = ' + str(y))
```



## Let's see why

```
def f(x):
    y = 1
    x = x + y
    print('x = ' = str(x))
    return x

x = 3
y = 2
z = f(x)
print('z = ' + str(z))
print('x = ' + str(x))
print('y = ' + str(y))
```




Now control reverts to the global environment, where the values of x, y and z are visible

## Another example

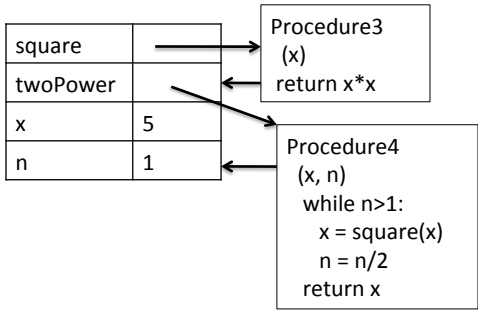
```
def square(x):
    return x*x

def twoPower(x, n):
    while n > 1:
        x = square(x)
        n = n/2
    return x
```

## Some observations

- Each function call creates a new environment, which scopes bindings of formal parameters and values, and of local variables (those created with assignments within body)
- Scoping often called static or lexical because scope within which variable has value is defined by extent of code boundaries

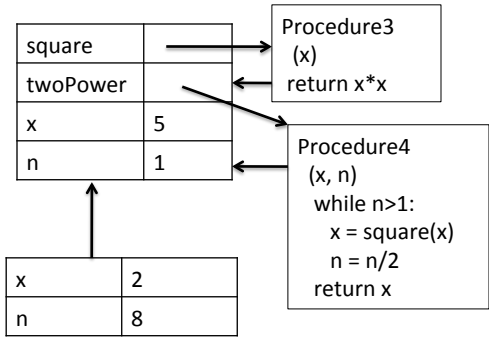
## Let's try it out

```
def square(x):
    return x*x

def twoPower(x, n):
    while n > 1:
        x = square(x)
        n = n/2
    return x

x = 5
n = 1
print(twoPower(2,8))
```



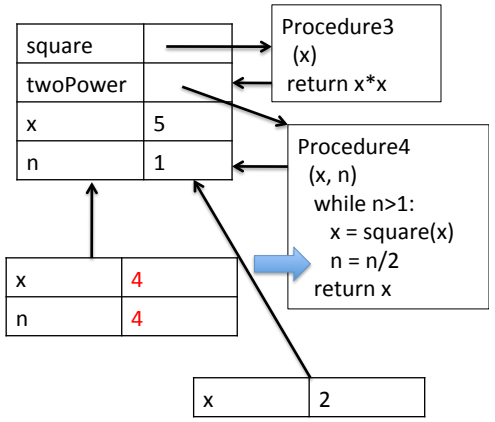
# Let's try it out

```
def square(x):
    return x*x

def twoPower(x, n):
    while n > 1:
        x = square(x)
        n = n/2
    return x

x = 5
n = 1
print(twoPower(2,8))
```



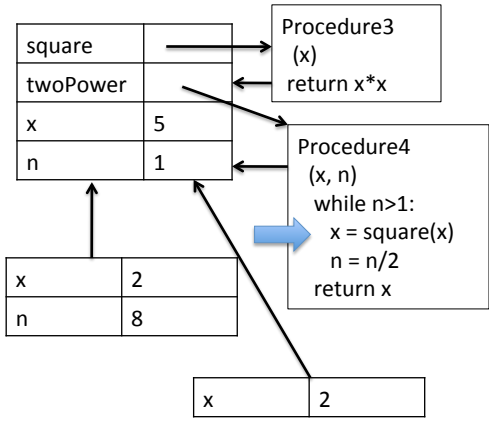

# Let's try it out

```
def square(x):
    return x*x

def twoPower(x, n):
    while n > 1:
        x = square(x)
        n = n/2
    return x

x = 5
n = 1
print(twoPower(2,8))
```



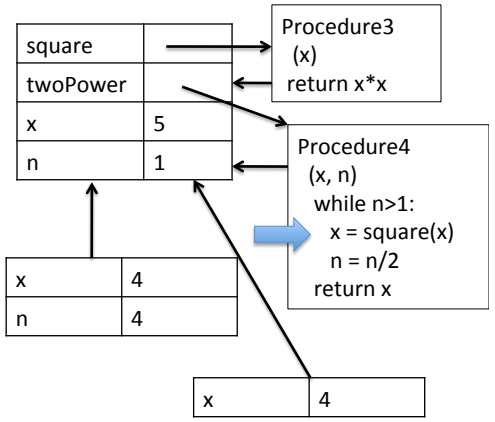

# Let's try it out

```
def square(x):
    return x*x

def twoPower(x, n):
    while n > 1:
        x = square(x)
        n = n/2
    return x

x = 5
n = 1
print(twoPower(2,8))
```



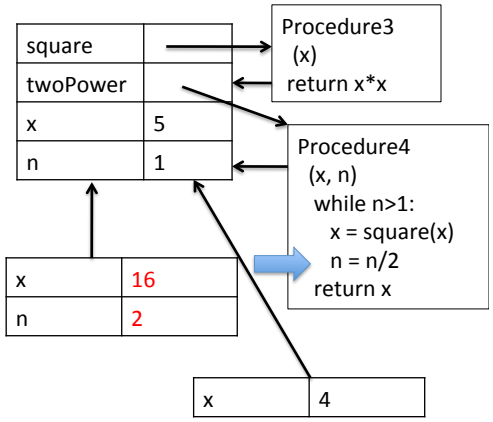

# Let's try it out

```
def square(x):
    return x*x

def twoPower(x, n):
    while n > 1:
        x = square(x)
        n = n/2
    return x

x = 5
n = 1
print(twoPower(2,8))
```



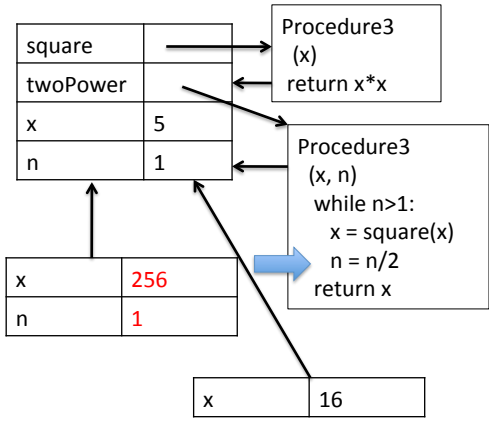
# Let's try it out

```
def square(x):
    return x*x

def twoPower(x, n):
    while n > 1:
        x = square(x)
        n = n/2
    return x

x = 5
n = 1
print(twoPower(2,8))
```



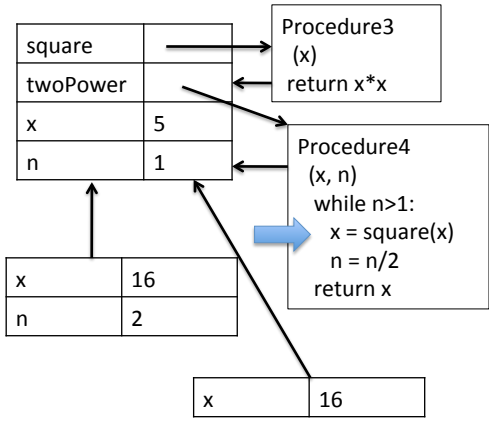

# Let's try it out

```
def square(x):
    return x*x

def twoPower(x, n):
    while n > 1:
        x = square(x)
        n = n/2
    return x

x = 5
n = 1
print(twoPower(2,8))
```



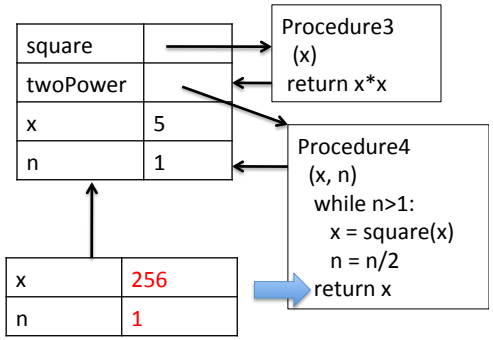

# Let's try it out

```
def square(x):
    return x*x

def twoPower(x, n):
    while n > 1:
        x = square(x)
        n = n/2
    return x

x = 5
n = 1
print(twoPower(2,8))
```




# Let's try it out

```
def square(x):
    return x*x

def twoPower(x, n):
    while n > 1:
        x = square(x)
        n = n/2
    return x

x = 5
n = 1
print(twoPower(2,8))
256
```



# Some observations

- Notice how each call to square created a new frame, with a local binding for x
- The value of x in the global environment was never confused with values within frames from function calls
- The value of x used by the call to square is different from the binding for x in the call to twoPower
- The rules we described can be followed mechanically to determine scoping of variables

# Capturing a more interesting example

```
def findRoot2(x, power, epsilon):
    if x < 0 and power%2 == 0:
        return None
    # can't find even powered root of
  negative number
    low = min(0, x)
    high = max(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
```

```
findRoot2(25.0, 2, .001)
4.99992370605

findRoot2(27.0, 3, .001)
2.99998855591

findRoot2(-27.0, 3, .001)
-2.99998855591
```

# Capturing a more interesting example

```
def findRoot1(x, power, epsilon):
    low = 0
    high = 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
```

```
findRoot1(25.0, 2, .001)
4.99992370605

findRoot1(27.0, 3, .001)
2.99998855591

findRoot1(-27.0, 3, .001)
```

Why does this fail on the third example?

# Capturing a more interesting example

```
def findRoot2(x, power, epsilon):
    if x < 0 and power%2 == 0:
        return None
    # can't find even powered root of
  negative number
    low = min(0, x)
    high = max(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
```

```
findRoot2(25.0, 2, .001)
4.99992370605

findRoot2(27.0, 3, .001)
2.99998855591

findRoot2(-27.0, 3, .001)
-2.99998855591

findRoot2(0.25, 2, .001)
```

Why does this fail on the fourth example?

# Think about our bisection search

- When we call with a fractional argument, like .25, we are searching


- Which means our first guess will be the average, or .125
  - Our original idea used the fact that the root of x was between 0 and x, but when x is fractional, the root is between x and 1

# Adding a specification

```
def findRoot3(x, power, epsilon):
    """x and epsilon int or float, power an int
           epsilon > 0 & power >=1
       returns a float y s.t. 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
    # can't find even powered root of negative number
    low = min(-1, x)
    high = max(1, 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
```

# Capturing a more interesting example

```
def findRoot3(x, power, epsilon):
    if x < 0 and power%2 == 0:
        return None
    # can't find even powered root of
  negative number
    low = min(-1, x)
    high = max(1, 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
```

```
findRoot3(25.0, 2, .001)
4.99992370605

findRoot3(27.0, 3, .001)
2.99998855591

findRoot3(-27.0, 3, .001)
-2.99998855591

findRoot3(0.25, 2, .001)
0.5

findRoot3(0.25, 2, .001)
-0.5
```

# Specifications

- Are a contract between implementer of function and user
  - Assumptions: conditions that must be met by users of function. Typically constraints on parameters, such as type, and sometimes acceptable ranges of values
  - Guarantees: Conditions that must be met by function, provided that it has been called in way that satisfies assumptions

# Functions close the loop

- Can now create new procedures and treat as if Python primitives
- Properties
  - Decomposition: Break problems into modules that are self-contained, and can be reused in other settings
  - Abstraction: Hide details. User need not know interior details, can just use as if a black box.

# Example

```
pi = 3.14159

def area(radius):
    return pi*(radius**2)

def circumference(radius):
    return 2*pi*radius

def sphereSurface(radius):
    return 4.0*area(radius)

def sphereVolume(radius):
    return (4.0/3.0)*pi*(radius**3)
```

Be sure to save this code in a file called `circle.py`

# Using functions in modules

- Modularity suggests grouping functions together that share common theme
- Place in a single .py file
- Use import command to access

# Example

```
import circle
pi = 3.0
print pi
print circle.pi
print circle.area(3)
print
  circle.circumference(3)
```

- Will result in

3.0 # value from local env

3.14159 # value from file

28.27431 # uses values from file

18.84954

- The . notation specifies context from which to read values

# Example

```
from circle import *
pi = 0.0
print pi
print area(3)
print circumference(3)
```

- Will result in

0.0 # value from local scope

28.27431 # uses values from file, because area not bound locally, but inherits from circle; however format allows reference as if in local scope

18.84954