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
```

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

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
```

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
```

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.