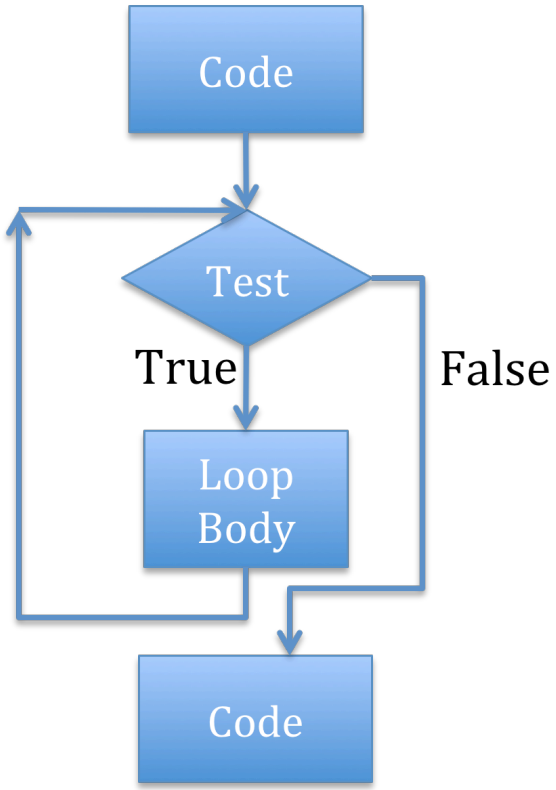
Iteration

- Need one more concept to be able to write programs of arbitrary complexity
  - Start with a test
  - If evaluates to `True`, then execute loop body once, and go back to reevaluate the test
  - Repeat until test evaluates to `False`, after which code following iteration statement is executed



An example

```
x = 3
ans = 0
itersLeft = x
while (itersLeft != 0):
    ans = ans + x
    itersLeft = itersLeft – 1
print(str(x) + '*' + str(x) + ' = ' + str(ans))
```

This code squares the value of x by repetitive addition.

Stepping through this code

```
x = 3
ans = 0
itersLeft = x
while (itersLeft != 0):
    ans = ans + x
    itersLeft = itersLeft – 1
print(str(x) + '*' + str(x) +
   ' = ' + str(ans))
```

x	ans	itersLeft
3	0	3

Some properties of iteration loops:

- need to set an iteration variable outside the loop
- need to test that variable to determine when done
- need to change that variable within the loop, in addition to other work

Iterative code

- Branching structures (conditionals) let us jump to different pieces of code based on a test
  - Programs are **constant time**
- Looping structures (e.g., while) let us repeat pieces of code until a condition is satisfied
  - Programs now take time that depends on values of variables, as well as length of program

Classes of algorithms

- Iterative algorithms allow us to do more complex things than simple arithmetic
- We can repeat a sequence of steps multiple times based on some decision; leads to new classes of algorithms
- One useful example are “guess and check” methods

Guess and check

- Remember our "declarative" definition of square root of `x`
- If we could guess possible values for square root (call it `g`), then can use definition to check if `g*g = x`
- We just need a good way to generate guesses

Finding a cube root of an integer

- One way to use this idea of generating guesses in order to find a cube root of `x` is to first try `0**3`, then `1**3`, then `2**3`, and so on
- Can stop when reach `k` such that `k**3 > x`
- Only a finite number of cases to try

Some code

```
x = int(raw_input('Enter an integer: '))
ans = 0
while ans**3 < x:
    ans = ans + 1
if ans**3 != x:
    print(str(x) + ' is not a perfect cube')
else:
    print('Cube root of ' + str(x) + ' is '
  + str(ans))
```

Extending scope

- Only works for positive integers
- Easy to fix by keeping track of sign, looking for solution to positive case

Some code

```
x = int(raw_input('Enter an integer: '))
ans = 0
while ans**3 < abs(x):
    ans = ans + 1
if ans**3 != abs(x):
    print(str(x) + ' is not a perfect cube')
else:
    if x < 0:
        ans = - ans
    print('Cube root of ' + str(x) + ' is '
  + str(ans))
```

Loop characteristics

- Need a loop variable
  - Initialized outside loop
  - Changes within loop
  - Test for termination depends on variable
- Useful to think about a **decrementing function**
  - Maps set of program variables into an integer
  - When loop is entered, value is non-negative
  - When value is <= 0, loop terminates, and
  - Value is decreased every time through loop
- Here we use `abs(x) – ans**3`

What happens if we miss a condition?

- Suppose we don’t initialize the variable?

- Suppose we don’t change the variable inside the loop?

Exhaustive enumeration

- Guess and check methods can work on problems with a finite number of possibilities
- Exhaustive enumeration is a good way to generate guesses in an organized manner

For loops

- `While` loops generally iterate over a sequence of choices (`ints` in cases we have seen)
- Python has a specialized mechanism for this case, called a `for` loop

```
for <identifier> in <sequence>:
    <code block>
```

For loops

- Identifier bound to first value in sequence
- Code block executed
- Identifier bound to next value
- Code block executed
- Continues until sequence exhausted or a **break** statement is executed
- To generate a sequence of integers, use
  - `range(n) = [0, 1, 2, 3, …, n-1]`
  - `range(m,n) = [m, m+1, …, n-1]`

A cleaned up cube root finder

```
x = int(raw_input('Enter an integer: '))
for ans in range(0, abs(x)+1):
    if ans**3 == abs(x):
        break
if ans**3 != abs(x):
    print(str(x) + ' is not a perfect
  cube')
else:
    if x < 0:
        ans = - ans
    print('Cube root of ' + str(x) + ' is
  ' + str(ans))
```

Dealing with floats

- Floats approximate real numbers, but useful to understand how
- Decimal number:
  - 302 = 3*10**2 + 0*10**1 + 2*10**0

Remember: ** is Python's exponentiation operator

- Binary number
  - 10011 = 1*2**4 + 0*2**3 + 0*2**2 + 1*2**1 + 1*2**0
  - (which in decimal is 16 + 2 + 1 = 19)
- Internally, computer represents numbers in binary

Converting decimal integer to binary

- Consider example of
  - x = 1*2**4 + 0*2**3 + 0*2**2 + 1*2**1 + 1*2**0
- If we take remainder relative to 2 (`x%2`) of this number, that gives us the last binary bit
- If we then divide x by 2 (`x/2`), all the bits get shifted left
  - x/2 = 1*2**3 + 0*2**2 + 0*2**1 + 1*2**0 = 1001
- Keep doing successive divisions; now remainder gets next bit, and so on
- Let's convert to binary form

Doing this in Python

```
if num < 0:
    isNeg = True
    num = abs(num)
else:
    isNeg = False
result = ''
if num == 0:
    result = '0'
while num > 2:
    result = str(num%2) + result
    num = num/2
if isNeg:
    result = '-' + result
```

So what about fractions?

- 3/8 = 0.375 = 3*10**(-1) + 7*10**(-2) + 5*10**(-3)
- So if we multiply by a power of 2 big enough to convert into a whole number, can then convert to binary, then divide by the same power of 2
- 0.375 * (2**3) = 3 (decimal)
- Convert 3 to binary (now 11)
- Divide by 2**3 (shift left) to get 0.011 (binary)

```
x = float(raw_input('Enter a decimal number between 0 and 1: '))

p = 0
while ((2**p)*x)%1 != 0:
    print('Remainder = ' + str((2**p)*x - int((2**p)*x)))
    p += 1

num = int(x*(2**p))

result = ''
if num == 0:
    result = '0'
while num > 0:
    result = str(num%2) + result
    num = num/2

for i in range(p - len(result)):
    result = '0' + result

result = result[0:-p] + '.' + result[-p:]
print('The binary representation of the decimal ' + str(x) + ' is
' + str(result))
```

Some implications

- If there is no integer p such that x*(2**p) is a whole number, then internal representation is always an approximation
- Suggest that testing equality of floats is not exact
  - Use abs(x-y) < 0.0001, rather than x == y
- Why does print(0.1) return 0.1, if not exact?
  - Because Python designers set it up this way to automatically round

Approximate solutions

- Suppose we now want to find the square root of any non-negative number?
- Can't guarantee exact answer, but just look for something close enough
- Start with exhaustive enumeration
  - Take small steps to generate guesses in order
  - Check to see if close enough

Example code

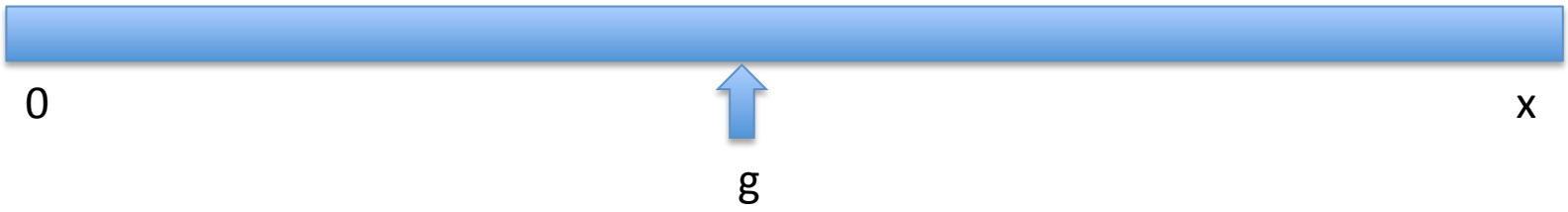
```
x = 25
epsilon = 0.01
step = epsilon**2
numGuesses = 0
ans = 0.0
while (abs(ans**2 - x)) >= epsilon and ans <= x:
    ans += step
    numGuesses += 1
print('numGuesses = ' + str(numGuesses))
if abs(ans**2-x) >= epsilon:
    print('Failed on square root of ' + str(x))
else:
    print(str(ans) + ' is close to the square root
  of ' + str(x))
```

Some observations

- Step could be any small number
  - If too small, takes a long time to find square root
  - If make too large, might skip over answer without getting close enough
- In general, will take x/step times through code to find solution
- Need a more efficient way to do this

Bisection search

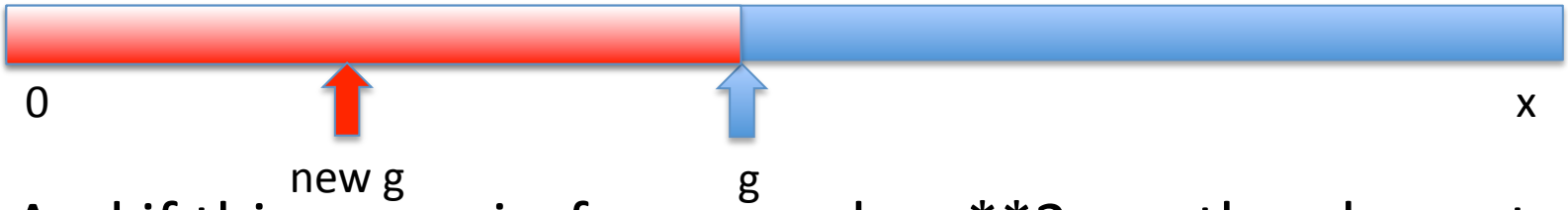
- We know that the square root of x lies between 0 and x, from mathematics
- Rather than exhaustively trying things starting at 0, suppose instead we pick a number in the middle of this range



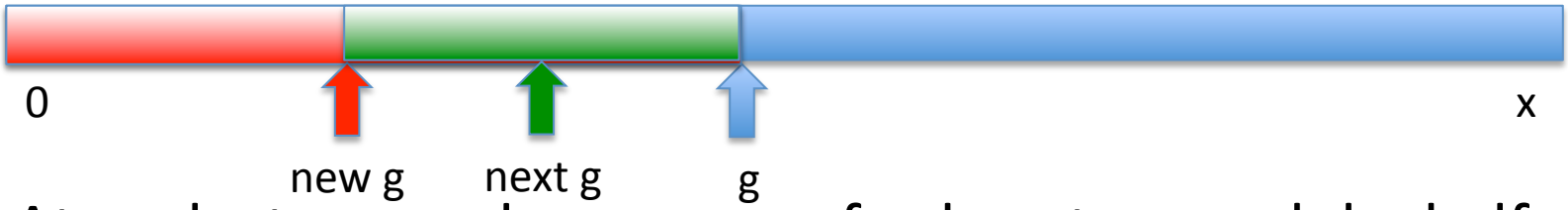
- If we are lucky, this answer is close enough

Bisection search

- If not close enough, is guess too big or too small?
- If g**2 > x, then know g is too big; but now search



- And if this new g is, for example, g**2 < x, then know too small; so now search



- At each stage, reduce range of values to search by half

Example of square root

```
x = 25
epsilon = 0.01
numGuesses = 0
low = 0.0
high = x
ans = (high + low)/2.0
while abs(ans**2 - x) >= epsilon:
    print('low = ' + str(low) + ' high = ' + str(high) + ' ans = ' + str(ans))
    numGuesses += 1
    if ans**2 < x:
        low = ans
    else:
        high = ans
    ans = (high + low)/2.0
print('numGuesses = ' + str(numGuesses))
print(str(ans) + ' is close to square root of ' + str(x))
```

Some observations

- Bisection search radically reduces computation time – being smart about generating guesses is important
- Should work well on problems with "ordering" property – value of function being solved varies monotonically with input value
  - Here function is g**2; which grows as g grows

Newton-Raphson

- General approximation algorithm to find roots of a polynomial in one variable

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$$

- Want to find r such that p(r) = 0
- For example, to find the square root of 24, find the root of $p(x) = x^2 - 24$
- Newton showed that if g is an approximation to the root, then

$$g - p(g)/p'(g)$$

is a better approximation; where p' is derivative of p

Newton-Raphson

- Simple case: $cx^2 + k$
- First derivative: $2cx$
- So if polynomial is $x^2 + k$, then derivative is $2x$
- Newton-Raphson says given a guess g for root, a better guess is

$$g - (g^2 - k)/2g$$

Newton-Raphson

- This gives us another way of generating guesses, which we can check; very efficient

```
epsilon = 0.01
y = 24.0
guess = y/2.0

while abs(guess*guess - y) >= epsilon:
    guess = guess - (((guess**2) - y)/(2*guess))
print('Square root of ' + str(y) + ' is about '
  + str(guess))
```

Iterative algorithms

- Guess and check methods build on reusing same code
  - Use a looping construct to generate guesses, then check and continue
- Generating guesses
  - Exhaustive enumeration
  - Bisection search
  - Newton-Raphson (for root finding)