LOOPS and STRINGS, GUESS-and-CHECK, APPROXIMATION, BISECTION

REVIEWING LOOPS

```
ans = 0
neg_flag = False
x = int(input("Enter an integer: "))
if x < 0:
    neg_flag = True
while ans**2 < x:
    ans = ans + 1
if ans**2 == x:
    print("Square root of", x, "is", ans)
else:
    print(x, "is not a perfect square")
    if neg_flag:
        print("Just checking... did you mean", -x, "?")
```

rewrite as ans += 1

guess and check ans

setting up the variable ans outside, and I'm changing the variable inside with a test that depends on it

REVIEWING STRINGS

- think of as a **sequence** of case sensitive characters
- can compare strings with ==, >, < etc. using lexicographic order
- `len()` is a function used to retrieve the **length** of the string in the parentheses
- square brackets used to perform **indexing** into a string to get the value at a certain index/position

```
s = "abc"
```

index: 0 1 2 ← indexing always starts at 0

`len(s)` → evaluates to 3

`s[0]` → evaluates to "a"

`s[1]` → evaluates to "b"

`s[3]` → trying to index out of bounds, error

STRINGS

- can **slice strings** using `[start:stop:step]`

  `s = "abcdefgh"`

  s[::-1] = 'hgfedcba'

  `s[::-1]` → evaluates to `"hgfedbca"`

  `s[3:6]` → evaluates to `"def"`

  `s[-1]` → evaluates to `"h"`

If unsure what some command does, try it out in your console!

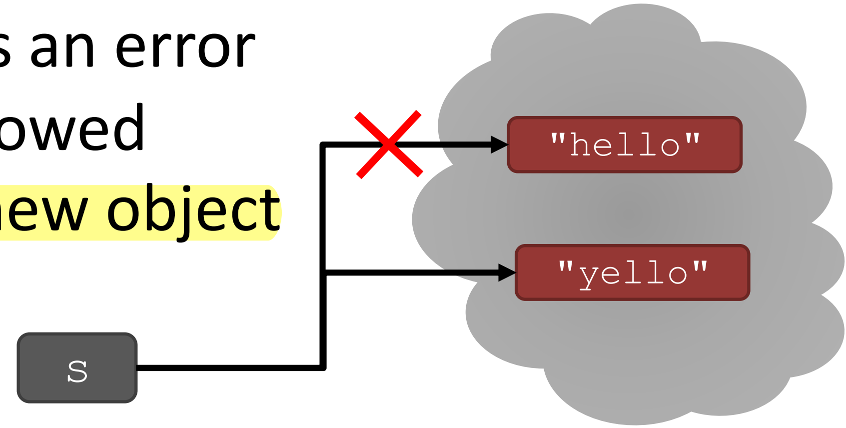
- strings are "**immutable**" – cannot be modified

  `s = "hello"`

  `s[0] = 'y'` → gives an error

  `s = 'y'+s[1:len(s)]` → is allowed

  `s` is a new object



FOR LOOPS RECAP

- `for` loops have a **loop variable** that iterates over a set of values

```
for var in range(4):
    <expressions>
```

  - `var` iterates over values 0,1,2,3
  - expressions inside loop executed with each value for `var`

```
for var in range(4,8):
    <expressions>
```

  - `var` iterates over values 4,5,6,7

- `range` is a way to iterate over numbers, but a for loop variable can iterate over any set of values, not just numbers!

can also iterate over string

STRINGS AND LOOPS

I can iterate over anything where
I can successively enumerate each
of the elements of that piece

```
s = "abcdefgh"
for index in range(len(s)):
    if s[index] == 'i' or s[index] == 'u':
        print("There is an i or u”)
```

string s is iterable

```
for char in s:
    if char == 'i' or char == 'u':
        print("There is an i or u”)
```

CODE EXAMPLE

```
an_letters = "aefhilmnorsxAEFHILMNORSX"

word = input("I will cheer for you! Enter a word: ")
times = int(input("Enthusiasm level (1-10): "))
i = 0

while i < len(word):
    char = word[i]
    if char in an_letters:
        print("Give me an " + char + "! " + char)
    else:
        print("Give me a  " + char + "! " + char)
    i += 1
print("What does that spell?")
for i in range(times):
    print(word, "!!!")
```

APPROXIMATE SOLUTIONS

- suppose we now want to find the 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

APPROXIMATE SOLUTIONS

- **good enough** solution
- start with a guess and increment by some **small value**
- `|guess`^{3}`|-cube <= epsilon`
  for some **small epsilon**

guess cubed, take the absolute value in case it's negative,
and look at the difference between that and cubed to see if it's less
than or equal to a small number
And if it is, I'm going to say I'm close enough
and I'm going to stop

If I make them really small,
I'll make sure I find a really
good guess but it's going to
slow the program down.

- decreasing increment size → slower program
- increasing epsilon → less accurate answer

easier to find an answer

APPROXIMATE SOLUTION – cube root

Step could be any small number.
If it's too small, it's going to take a long time and many guesses
if it's too large and I'm not careful,
it could skip over the answer without getting close enough.
In general, it's going to take x divided by step number times
through the code to find a solution

```
cube = 27
epsilon = 0.01
guess = 0.0
increment = 0.0001
num_guesses = 0
while abs(guess**3 - cube) >= epsilon  and guess <= cube :
    guess += increment
    num_guesses += 1
print('num_guesses =', num_guesses)
if abs(guess**3 - cube) >= epsilon:
    print('Failed on cube root of', cube)
else:
    print(guess, 'is close to the cube root of', cube)
```

Cube is value for the thing I'm trying to find the cube of.
Epsilon is going to be something that tells me how close I want to get to the answer.
Guess is where I'm going to start.
And increments, the size in which I'm going to increase my guess as I move along.

keep track of how many times do I go through the loop as I do this

without this, if I've gone right past the cube root, it would still run in an infinite loop

Some observations

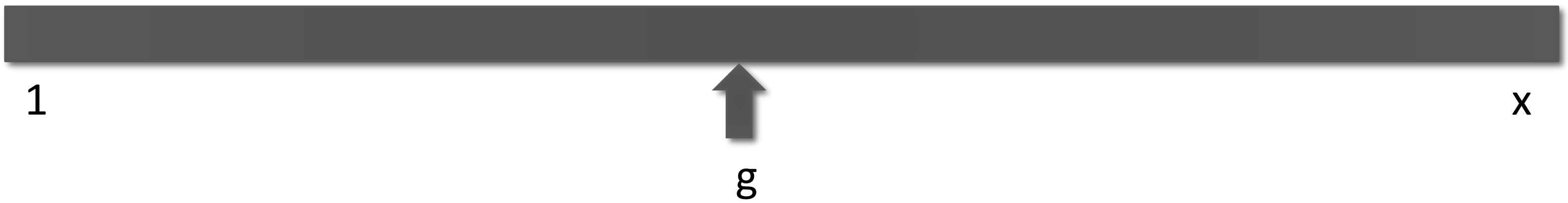
- Step could be any small number
  - If too small, takes a long time to find square root
  - If too large, might skip over answer without getting close enough
- In general, will take x/step times through code to find solution
- Need a more efficient way to do this

-> search for John Harvard in dictionary

This is a powerful tool, that idea of bisection search,
throwing away half the possible values at every stage,
let's me very quickly zero in (compared to exhaustive enumeration!)

BISECTION SEARCH

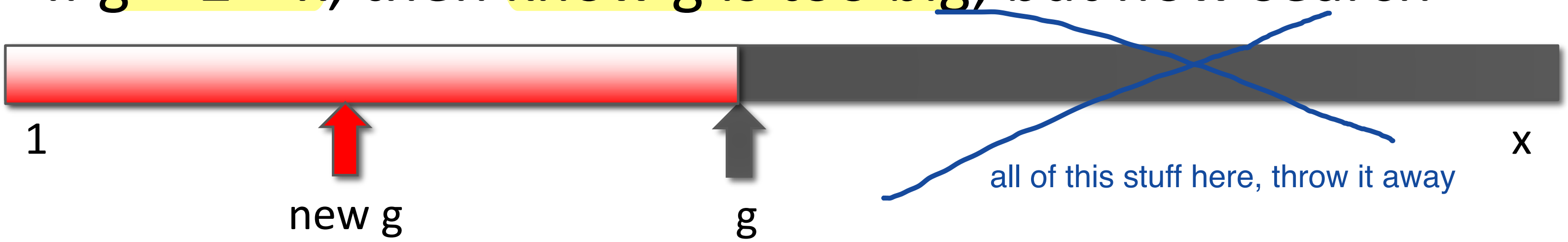
- We know that the square root of x lies between 1 and x, from mathematics
- Rather than exhaustively trying things starting at 1, suppose instead we pick a number in the middle of this range



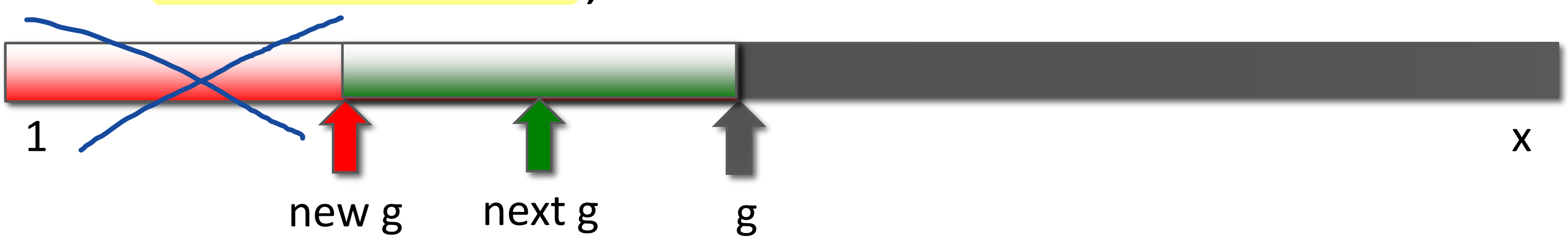
- If we are lucky, this answer is close enough

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, for example, this new g is such that g**2 < x, then know too small; so now search



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

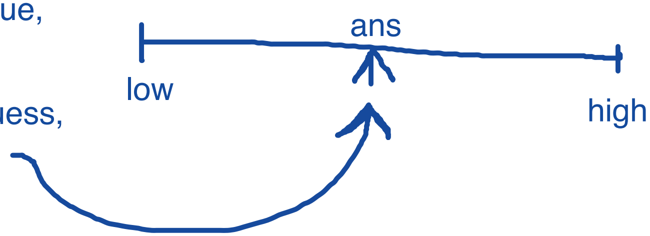
EXAMPLE OF SQUARE ROOT

```
x = 25
epsilon = 0.01
numGuesses = 0
low = 1.0
high = x
ans = (high + low)/2.0
```

I'm going to look for square root of
I'm going to use to tell how close I am
'm also going to keep track of how often do I actually need to make guesses

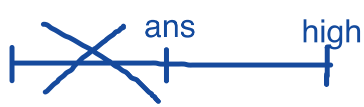
I need to set a low value and a high value,
the range in which I'm going to look

And then I'm going to make an initial guess,
which is halfway in between

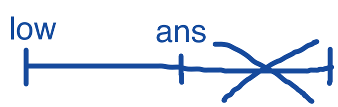


-> I'm not close enough

```
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))
```



if answer is too small, that is,
answer squared is less than x,
then I can change the low part of the range to move up to answer



if answer is too big, that is,
answer squared is more than x,
then I can change the high part of the range to move up to answer

BISECTION SEARCH – cube root

only one more step needed than with square root

```
cube = 27
epsilon = 0.01
num_guesses = 0
low = 1
high = cube
guess = (high + low)/2.0
while abs(guess**3 - cube) >= epsilon:
    if guess**3 < cube :
        low = guess
    else:
        high = guess
    guess = (high + low)/2.0
    num_guesses += 1
print('num_guesses =', num_guesses)
print(guess, 'is close to the cube root of', cube)
```

BISECTION SEARCH CONVERGENCE

- search space
  - first guess: N/2
  - second guess: N/4
  - gth guess: $N/2^g$

I don't take a linear time, I actually take less than linear time to get there (amount of time grows as the log of n)

- guess converges on the order of $\log_2 N$ steps
- bisection search works when value of function varies monotonically with input
- code as shown only works for positive cubes > 1 – why?
- challenges → modify to work with negative cubes!
  → modify to work with x < 1!

for x<1:
x=0.5
3√0.5 = 0.793700526

for x<0:
x=--8
3√-8 = -2

x < 1

- if x < 1, search space is 0 to x but cube root is greater than x and less than 1
- modify the code to choose the search space depending on value of x

Start with a basic set of code, check to see what it runs on,
and then decide if I wanted to use the same code,
how could small changes have it run
on other kinds of solutions?

SOME OBSERVATIONS

I went from 30,000 times through a loop (exhaustive enumeration)
to 10 or 15 times through a loop (bisection search)

- Bisection search radically reduces computation time – being smart about generating guesses is important

it increases as the input value increases.

- 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

G squared has that property.
As I change g, it grows.
G squared grows as g grows.
G cubed, same kind of idea.

** Your initial endpoints should be 0 and 100. Do not optimize your subsequent endpoints by making them be the halfway point plus or minus 1. Rather, just make them be the halfway point.

Python Trick: Printing on the same line

Try the following in your console:

```
# Notice how if we have two print statements
print("Hi")
print("there")

# The output will have each string on a separate line:
Hi
there

# Now try adding the following:
print("Hi",end='')
print("there")
print("Hi",end='*')
print("there")

# The output will place the subsequent string on the same line
# and will connect the two prints with the character given by end
Hithere
Hi*there
```

Test Cases to Test Your Code With. Be sure to test these on your own machine - and that you get the same output! - before running your code on this webpage!

Test case 1. Secret guess = 42

```
Please think of a number between 0 and 100!
Is your secret number 50?
```

DEALING WITH `float`'s

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

position

ten possible digitis from 0 to 9

- Binary number

how the computer store things, are actually represented the same way, but now in terms of powers of 2. (just two possible digits: 0 and 1)

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

CONVERTING DECIMAL INTEGER TO BINARY

- Consider example of
  - $x = 1*2^4 + 0*2^3 + 0*2^2 + 1*2^1 + 1*2^0 = 10011$

19 % 2 = 1

- 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 right
  - $x//2 = 1*2^3 + 0*2^2 + 0*2^1 + 1*2^0 = 1001$
- Keep doing successive divisions; now remainder gets next bit, and so on
- Let's us 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 > 0:
    result = str(num%2) + result
    num = num//2
if isNeg:
    result = '-' + result
```

If the number is less than 0,
I'm going to put on a flag that says it's negative.

use the positive version of it

I'm going to simply accumulate those results

If the number is equal to 0, it's just 0

Otherwise, as long as the number is greater than 0,
I'm going to do remainder, add that in,
because that's the next bit into result,
do the division to shift it to the right, and keep going.

In decimal (our normal number system), we multiply by 10 to shift left and divide by 10 to shift right (like 123 / 10 = 12.3).
In binary, it's the same but with 2! It's like how dividing by 10**3 (1000) shifts 3 right in decimal

WHAT ABOUT FRACTIONS?

- $3/8 = 0.375 = 3*10^{-1} + 7*10^{-2} + \underline{5}*10^{-3}$

  position

  number of possible digitis

  placeholder

- So if we multiply by a power of 2 big enough to convert into a whole number, can then convert to binary, and then divide by the same power of 2
- 0.375 * (2**3) = 3 (decimal)

  I'm simply taking powers of 2
  because that's moving the placeholder, for the decimal point,
  or I should say the binary point in a binary representation.

- Convert 3 to binary (now 11)
- Divide by 2**3 (shift right) to get 0.011 (binary)

I'm using something I did for one computation,
but converting another problem into the same problem.
In this case, given a fraction, I'm
saying find a power of 2 that shifts it into an integer,
use the same machinery, and then shift it back.

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

p is the integer of how many binary shifts to the right we need to do.
It's how many digits there will be after the decimal point

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

loop to try out different p. -> so that x multiplied by this power p of 2 is big enough to convert into a whole number (whole number%1 ==0)

if p is not yet big enough, %1 !=0
increase p and try again

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

multiply by the power p (weve found in our loop) of 2 big enough to convert into a whole number

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

convert decimal whole number into binary

difference p - len(result) is how many extra shifts we're doing after we've already shifted the decimal point past the digits we have already

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

decimal number 123. You want to shift it 5 to the right. This would go (Shift 1) 12.3, (Shift 2) 1.23, (Shift 3) .123. How do we keep shifting it now that we've run out of digits? We add 0s in front. In this case, we need two zeros to reach 0.00123. We needed (shifts) - (original number of digits) extra 0s. That's what p - len(result) is

part of the string before our fractional digits (which will be nothing unless the original number had a non-fractional part)

```python
result = result[0:-p] + '.' + result[-p:]
```

puts the decimal point in the string

```python
print('The binary representation of the decimal ' + str(x) + ' is
' + str(result))
```

result is a string holding our current value in binary.

if you're doing this with numbers in the range 0 < x < 1,
you could replace that last line with result = '.' + 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) < some small number, 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

The internal representation of the decimal number 1/10 = 0.1 requires an infinite number of digits.
floats stored in binary are always approximate unless they can be expressed as the sum of powers of 2.

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

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

it lets me not only find
cube roots and square roots, it lets me find
the solution to any equation very quickly

NEWTON-RAPHSON

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

```
epsilon = 0.01
```
decide how close I am to an answer

```
y = 24.0
guess = y/2.0
```
initial guess

```
numGuesses = 0

while abs(guess*guess - y) >= epsilon:
```
As long as guess squared minus y, the absolute value of that is too big, I'm going to keep going

```
    numGuesses += 1
    guess = guess - (((guess**2) - y)/(2*guess))
```
equation from Newton-Raphson: take g squared minus y divided by 2 times g (guess), subtract that off of guess, and go back around with a better approximation

```
print('numGuesses = ' + str(numGuesses))
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)