# 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`

# REVIEWING STRINGS

- think of as a **sequence** of case sensitive characters
- can compare strings with ==, >, < etc.
- `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]`

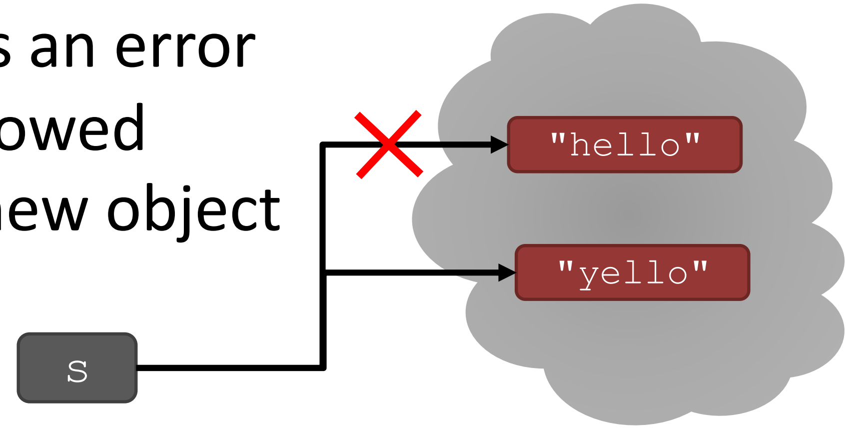
```
s = "abcdefgh"
s[::-1]     → evaluates to "hgfedbca"
s[3:6]      → evaluates to "def"
s[-1]       → evaluates to "h"
```

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!

# STRINGS AND LOOPS

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

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**

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

# APPROXIMATE SOLUTION – cube root

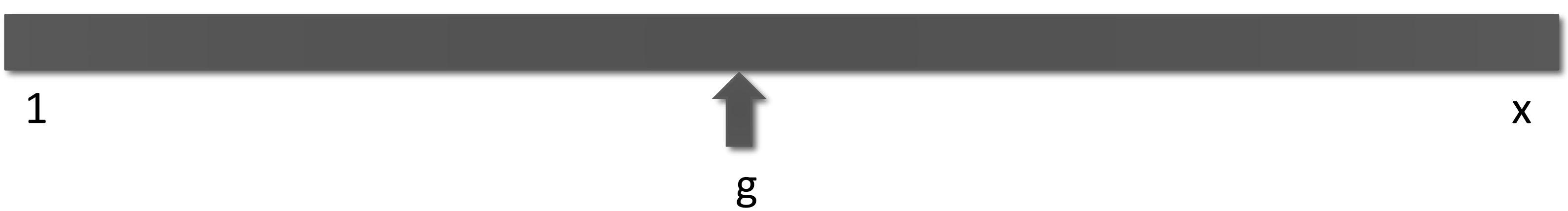
```
cube = 27
epsilon = 0.01
guess = 0.0
increment = 0.0001
num_guesses = 0
while abs(guess**3 - cube) >= epsilon  and guess <= cube :
    guess += increment
    num_guesses += 1
print('num_guesses =', num_guesses)
if abs(guess**3 - cube) >= epsilon:
    print('Failed on cube root of', cube)
else:
 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

# 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

# BISECTION SEARCH

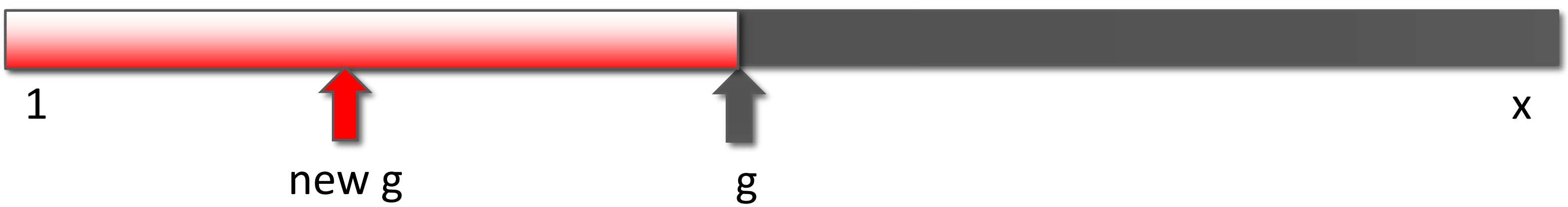
- We know that the square root of x lies between 1 and x, from mathematics
- Rather than exhaustively trying things starting at 1, suppose instead we pick a number in the middle of this range



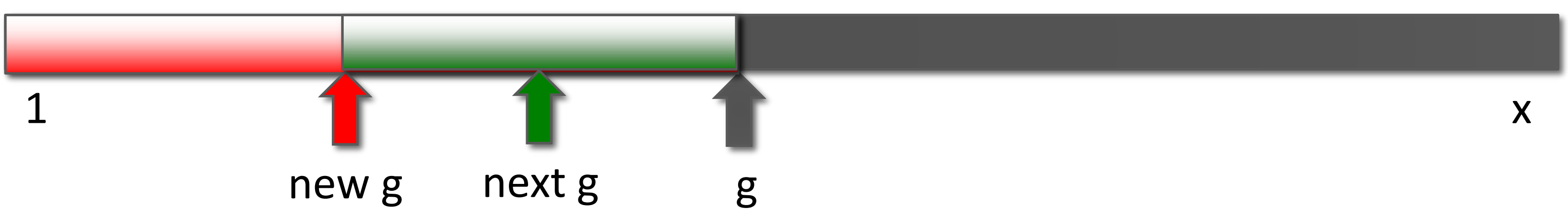

- If we are lucky, this answer is close enough

# 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

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))
```

# BISECTION SEARCH – cube 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: $N/2^g$
- 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!

# $x < 1$

- 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

# 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

# DEALING WITH `float`'s

- Floats approximate real numbers, but useful to understand how
- Decimal number:
  - $302 = 3*10^2 + 0*10^1 + 2*10^0$
- 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
  - $x = 1*2^4 + 0*2^3 + 0*2^2 + 1*2^1 + 1*2^0 = 10011$
- 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
```

# 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, and 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 right) to get 0.011 (binary)

```python
x = float(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) < 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

# NEWTON-RAPHSON

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

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

- 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
numGuesses = 0

while abs(guess*guess - y) >= epsilon:
    numGuesses += 1
    guess = guess - (((guess**2) - y)/(2*guess))
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)