LOOPS OVER STRINGS, GUESS-and-CHECK, BINARY

(download slides and .py files to follow along)

6.100L Lecture 4

Ana Bell

LAST TIME

- Looping mechanisms
  - `while` and `for` loops
- While loops
  - Loop as long as a **condition is true**
  - Need to make sure you don't enter an **infinite loop**
- For loops
  - Loop variable takes on values in a sequence, one at a time
  - Can loop over **ranges** of numbers
  - Will soon see many other things are easy to loop over

`break` STATEMENT

- Immediately exits whatever loop it is in
- Skips remaining expressions in code block
- **Exits only innermost loop**!

```
while <condition_1>:
    while <condition_2>:
        <expression_a>
        break
        <expression_b>
    <expression_c>
```

Evaluated when <condition_1> and <condition_2> are True

Never evaluated (don't write code like this)

Evaluated when <condition_1> is True

`break` STATEMENT

```
mysum = 0
for i in range(5, 11, 2):
    mysum += i
    if mysum == 5:
        break
        mysum += 1
print(mysum)
```

- What happens in this program?
- [Python Tutor LINK]()

YOU TRY IT!

- Write code that loops a `for` loop over some range and prints how many even numbers are in that range. Try it with:
  - `range(5)`
  - `range(10)`
  - `range(2,9,3)`
  - `range(-4,6,2)`
  - `range(5,6)`

STRINGS and LOOPS

- Code to check for letter i or u in a string.
- All 3 do the same thing

```
s = "demo loops - fruit loops"
for index in range(len(s)):
    if s[index] == 'i' or s[index] == 'u':
        print("There is an i or u")
```

Uses range to iterate through index of s

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

Iterates through characters of s directly

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

Iterates through characters of s directly (most "pythonic")

BIG IDEA

The sequence of values in a `for` loop isn't limited to numbers

ROBOT CHEERLEADERS

```
an_letters = "aefhilmnorsxAEFHILMNORSX"

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

for c in word:
    if c in an_letters:
        print(f'Give me an {c}: {c}')
    else:
        print(f'Give me a {c}: {c}')
print("What's that spell?")
for i in range(times):
    print(word, '!!!')
```

c is a loop variable whose value is each letter that the user gave

i is a loop variable whose value is 0 through times-1, one at a time

YOU TRY IT!

- Assume you are given a string of lowercase letters in variable s. Count how many unique letters there are in the string. For example, if

```
s = "abca"
```

Then your code prints 3.

HINT:
Go through each character in s.
Keep track of ones you've seen in a string variable.
Add characters from s to the seen string variable if they are not already a character in that seen variable.

SUMMARY SO FAR

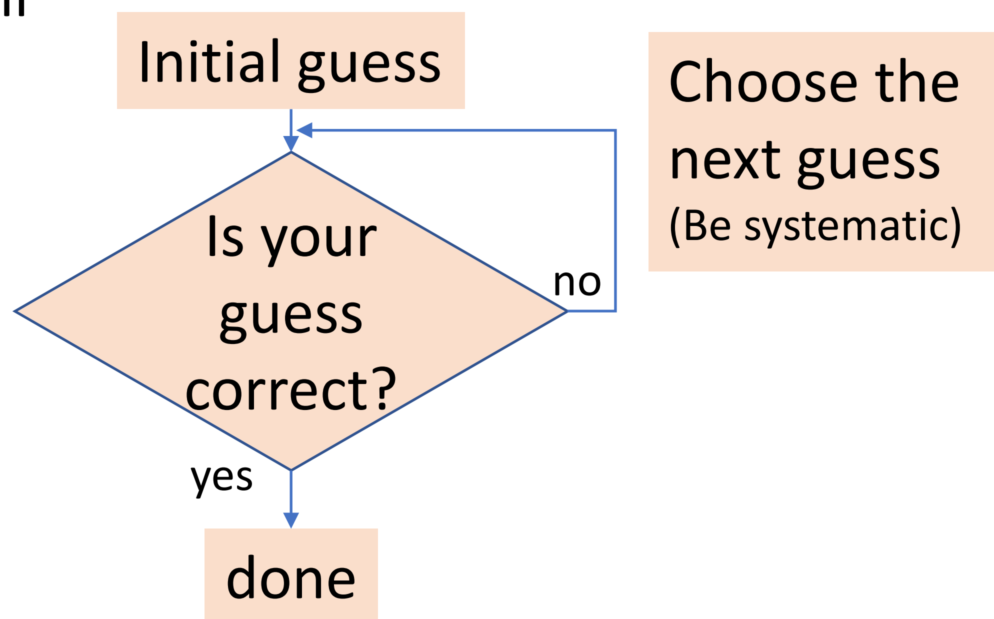
- Objects have **types**
- Expressions are **evaluated to one value**, and bound to a variable name
- Branching
  - if, else, elif
  - Program executes **one set of code or another**
- Looping mechanisms
  - `while` and `for` loops
  - Code executes repeatedly **while some condition is true**
  - Code executes repeatedly **for all values in a sequence**

THAT IS ALL YOU NEED TO IMPLEMENT ALGORITHMS

GUESS-and-CHECK

GUESS-and-CHECK

- Process called **exhaustive enumeration**
- Applies to a problem where …
  - You are able to **guess a value** for solution
  - You are able to **check if the solution is correct**
- You can **keep guessing** until
  - Find solution or
  - Have guessed all values



GUESS-and-CHECK SQUARE ROOT

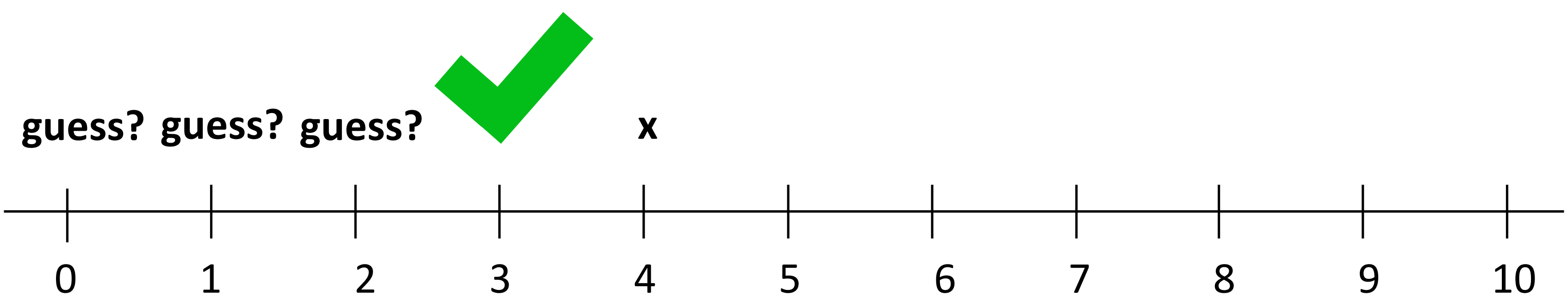
- Basic idea:
  - Given an `int`, call it `x`, want to see if there is another `int` which is its square root
  - Start with a `guess` and check if it is the right answer

GUESS-and-CHECK SQUARE ROOT

- Basic idea:
  - Given an `int`, call it `x`, want to see if there is another `int` which is its square root
  - Start with a `guess` and check if it is the right answer
  - To be **systematic**, start with `guess` = 0, then 1, then 2, etc

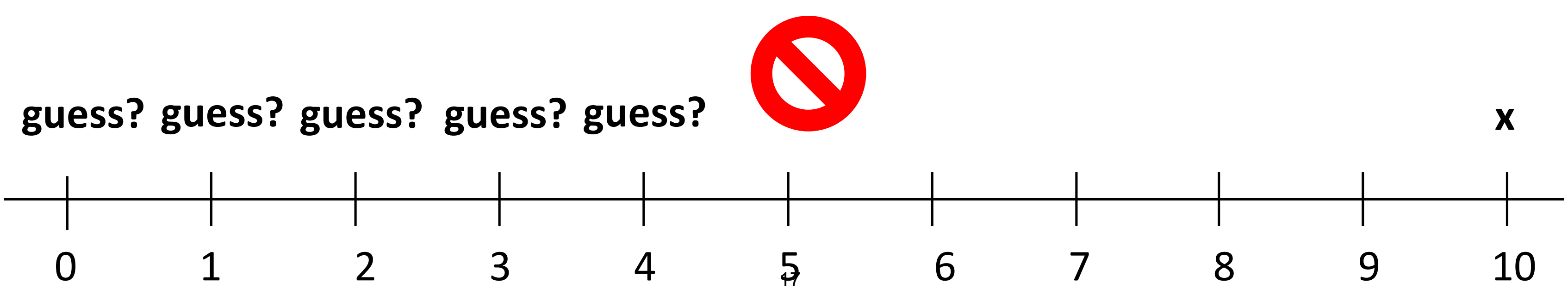
GUESS-and-CHECK SQUARE ROOT

- Basic idea:
  - Given an `int`, call it `x`, want to see if there is another `int` which is its square root
  - Start with a `guess` and check if it is the right answer
  - To be **systematic**, start with `guess` = 0, then 1, then 2, etc
- If `x` is a **perfect square**, we will **eventually find its root** and can stop (look at guess squared)



GUESS-and-CHECK SQUARE ROOT

- Basic idea:
  - Given an `int`, call it `x`, want to see if there is another `int` which is its square root
  - Start with a `guess` and check if it is the right answer
  - To be **systematic**, start with `guess` = 0, then 1, then 2, etc
- But what if `x` is **not a perfect square**?
  - Need to know when to stop
  - **Use algebra** – if `guess` squared is bigger than `x`, then can stop



GUESS-and-CHECK SQUARE ROOT with while loop

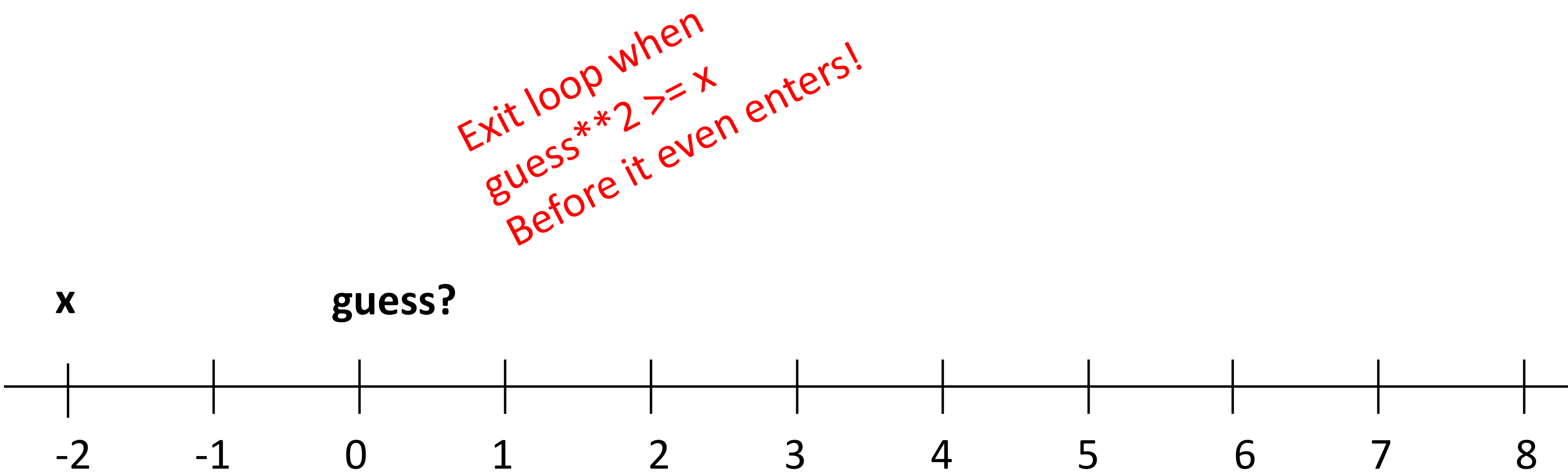
```
guess = 0
x = int(input("Enter an integer: "))
while guess**2 < x:
    guess = guess + 1
if guess**2 == x:
    print("Square root of", x, "is", guess)
else:
    print(x, "is not a perfect square")
```

Exit loop when guess**2 >= x

Check why you exited the loop

GUESS-and-CHECK SQUARE ROOT

- Does this work for any integer value of `x`?
- What if `x` is negative?
  - `while` loop immediately terminates
- Could **check for negative input**, and handle differently



GUESS-and-CHECK SQUARE ROOT with while loop

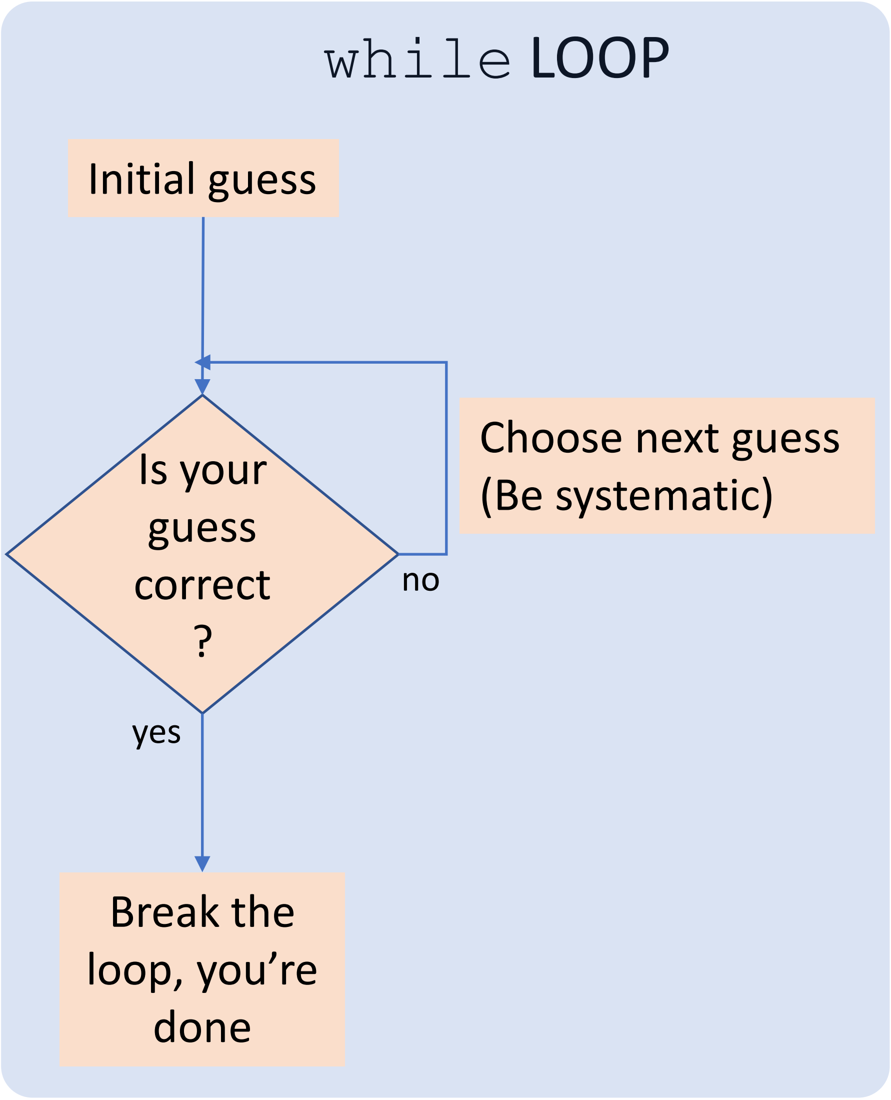
```
guess = 0
neg_flag = False
x = int(input("Enter a positive integer: "))
if x < 0:
    neg_flag = True
while guess**2 < x:
    guess = guess + 1
if guess**2 == x:
    print("Square root of", x, "is", guess)
else:
    print(x, "is not a perfect square")
    if neg_flag:
        print("Just checking...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

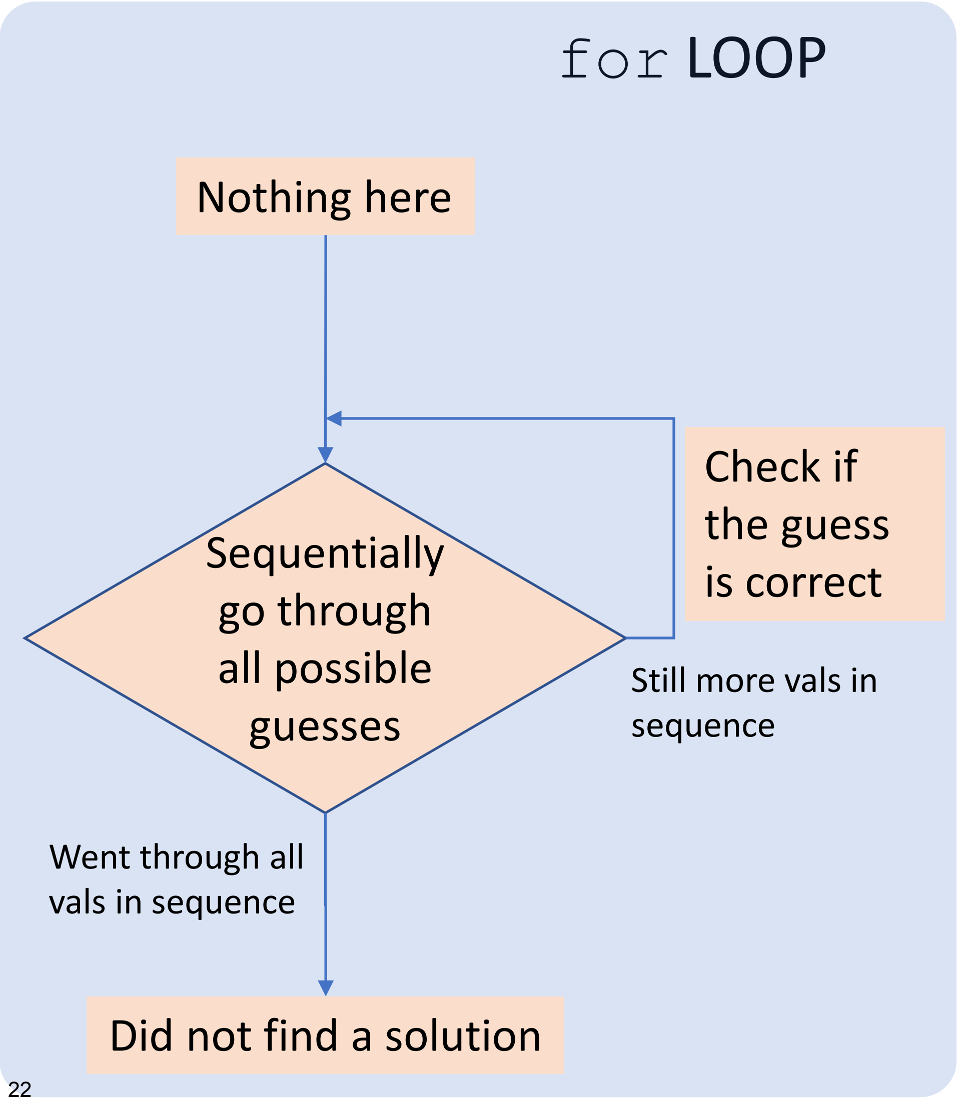
BIG IDEA

Guess-and-check can’t test an infinite number of values

You have to stop at some point!

GUESS-and-CHECK COMPARED





YOU TRY IT!

- Hardcode a number as a secret number.
- Write a program that checks through all the numbers from 1 to 10 and prints the secret value if it's in that range. **If it's not found, it doesn't print anything.**
- How does the program look if I change the requirement to be: **If it's not found, prints that it didn't find it.**

YOU TRY IT!

- Compare the two codes that:
  - Hardcode a number as a secret number.
  - Checks through all the numbers from 1 to 10 and prints the secret value if it's in that range.

If it's not found, it **doesn't print anything**.

Answer:

```
secret = 7

for i in range(1,11):
    if i == secret:
        print("yes, it's", i)
```

If it's not found, **prints that it didn't find it**.

Answer:

```
secret = 7
found = False
for i in range(1,11):
    if i == secret:
        print("yes, it's", i)
        found = True
if not found:
    print("not found")
```

BIG IDEA

Booleans can be used as signals that something happened

We call them Boolean flags.

`while` LOOP or `for` LOOP?

- Already saw that code looks **cleaner when iterating over sequences** of values (i.e. using a `for` loop)
  - Don't set up the iterant yourself as with a while loop
  - Less likely to introduce errors
- Consider an example that uses a `for` loop and an explicit `range` of values

GUESS-and-CHECK CUBE ROOT: POSITIVE CUBES

```
cube = int(input("Enter an integer: "))

for guess in range(cube+1):
    if guess**3 == cube:
        print("Cube root of", cube, "is", guess)
```

Want to include cube when cube is 1

GUESS-and-CHECK CUBE ROOT: POSITIVE and NEGATIVE CUBES

```
cube = int(input("Enter an integer: "))

for guess in range(abs(cube)+1):
    if guess**3 == abs(cube):
        if cube < 0:
            guess = -guess
        print("Cube root of "+str(cube)+" is "+str(guess))
```

Assume it's positive

Deal with negative cube here

GUESS-and-CHECK CUBE ROOT: JUST a LITTLE FASTER

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

Terminate search once know you have passed possible answer

Check why you exited the loop and decide if the guess is not a perfect cube

ANOTHER EXAMPLE

- Remember those word problems from your childhood?
- For example:
  - Alyssa, Ben, and Cindy are selling tickets to a fundraiser
  - Ben sells 2 fewer than Alyssa
  - Cindy sells twice as many as Alyssa
  - 10 total tickets were sold by the three people
  - How many did Alyssa sell?
- Could solve this algebraically, but we can also use guess-and-check

GUESS-and-CHECK with WORD PROBLEMS

Check all possible values

For each value of alyssa, check all possible values

For each pair of alyssa and ben, check all possible values

```
for alyssa in range(11):
    for ben in range(11):
        for cindy in range(11):
            total = (alyssa + ben + cindy == 10)
            two_less = (ben == alyssa-2)
            twice = (cindy == 2*alyssa)
            if total and two_less and twice:
                print(f"Alyssa sold {alyssa} tickets")
                print(f"Ben sold {ben} tickets")
                print(f"Cindy sold {cindy} tickets")
```

3 Booleans for our word problem equations

Solution found when all 3 hold

EXAMPLE WITH BIGGER NUMBERS

- With bigger numbers, nesting loops is slow!
- For example:
  - Alyssa, Ben, and Cindy are selling tickets to a fundraiser
  - Ben sells **20** fewer than Alyssa
  - Cindy sells **twice** as many as Alyssa
  - **1000** total tickets were sold by the three people
  - How many did Alyssa sell?
  - The previous code won't end in a reasonable time
- Instead, loop over one variable and code the equations directly

MORE EFFICIENT SOLUTION

One loop over one variable

Replace loops with direct calculation for other 2 values/people

Last condition

```
for alyssa in range(1001):
    ben = max(alyssa - 20, 0)
    cindy = alyssa * 2
    if ben + cindy + alyssa == 1000:
        print("Alyssa sold " + str(alyssa) + " tickets")
        print("Ben sold " + str(ben) + " tickets")
        print("Cindy sold " + str(cindy) + " tickets")
```

BIG IDEA

You can apply computation to many problems!

BINARY NUMBERS

NUMBERS in PYTHON

- **int**
  - integers, like the ones you learned about in elementary school
- **float**
  - reals, like the ones you learned about in middle school

OUR MOTIVATION - keep this in mind for the next few slides

```
x = 0
for i in range(10):
    x += 0.1
print(x == 1)
print(x, '==', 10*0.1)
```

Note: x += 0.1 is the same as x = x + 0.1

0.9999999999999999 == 1.0

BIG IDEA

Operations on some floats introduces a very small error.

The small error can have a big effect if operations are done many times!

A CLOSER LOOK AT FLOATS

- Python (and every other programming language) uses "floating point" to **approximate real numbers**
- The term "floating point" refers to the way these numbers are stored in computer
- Approximation usually doesn't matter
  - But it does for us!
  - Let's see why…

FLOATING POINT REPRESENTATION

- Depends on computer hardware, not programming language implementation
- Key things to understand
  - Numbers (and everything else) are represented as a **sequence of bits** (0 or 1).
  - When **we** write numbers down, the notation uses base 10.
    - 0.1 stands for the rational number 1/10
  - This produces **cognitive dissonance** – and it will influence how we write code

WHY BINARY?
HARDWARE IMPLEMENTATION

- Easy to implement in hardware—build components that can be in **one** of **two** states
- Computer hardware is built around methods that can efficiently store information as 0's or 1's and do arithmetic with this rep
  - a voltage is "high" or "low"   a magnetic spin is "up" or "down"
- Fine for integer arithmetic, but what about numbers with fractional parts (floats)?

BINARY NUMBERS

- Base 10 representation of an integer
  - sum of powers of 10, scaled by integers from 0 to 9

$1507 = 1*10^3 + 5*10^2 + 0*10^1 + 7*10^0$

$= 1000 + 500 + 7$

- Binary representation is same idea in base 2
  - sum of powers of 2, scaled by integers from 0 to 1
- $1507_{10} = 1*2^{10} + 1*2^8 + 1*2^7 + 1*2^6 + 1*2^5 + 1*2^1 + 1*2^0$

$= \boxed{1024} + 256 + 128 + 64 + 32 + 2 + 1$

$= 2^{10} + 2^8 + 2^7 + 2^6 + 2^5 + 2^1 + 2^0$

$= 10111100011_2$

Highest power of 2 to get us closest without going over to 1507

CONVERTING DECIMAL INTEGER TO BINARY

- We input integers in decimal, computer needs to convert to binary
- Consider example of
  - $x = 19_{10} = 1*2^4 + 0*2^3 + 0*2^2 + 1*2^1 + 1*2^0 = \boxed{1001}1$
- If we take **remainder of x relative to 2** `(x%2)`, that gives us the last binary bit
- If we then **integer 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 convert to binary form

DOING THIS in PYTHON for POSITIVE NUMBERS

Python Tutor LINK

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

DOING this in PYTHON and HANDLING NEGATIVE NUMBERS

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

Set a negative flag and handle it

SUMMARY

- Loops can iterate over any sequence of values:
  - range for numbers
  - A string
- Guess-and-check provides a **simple algorithm** for solving problems
  - When set of **potential solutions is enumerable**, exhaustive enumeration guaranteed to work (eventually)
- Binary numbers help us understand how the machine works
  - Converting to binary will help us understand how decimal numbers are stored
  - Important for the next algorithm we will see