Approximation

Recap

Floating point numbers can be tricky sometimes because of how computers work

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

Exercise

In codechum, there is an exercise

Assume you are given a string variable named `my_str`. Write code that prints out a new string containing the even indexed characters of `my_str`. For example, if `my_str` is "abcdef", the code should print "ace".

Because `abcdef` has `a` at index 0, `c` at index 2, `e` at index 4, etc.

Notes:

- to index a string, use `string or string variable[index]`
- you can make an empty string with `string variable name = ''`
- and you can add letters to that string through `string variable name += 'letter'` or `string variable name = string variable name + 'letter'`
- range has three parameters, `range(start, stop, step)`

Decimal to Binary

As a reminder

$$19_{10} = 1 * 10^1 + 9 * 10^0 = 1 * 2^4 + 0 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0 = 10011$$

And if we take the remainder of x `x % 2`, that gives us ___

Then if we integer divide x by 2 `x // 2`, all the bits shift right

Do this again, and again, until `x is 0`, or while `x > 0`

Decimal to Binary

```python
num = 1507
result = ''

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

Decimal to Binary

```
if num < 0:
    is_negative = True
    num = abs(num)
else:
    is_negative = False

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

if is_negative:
    result = "-" + result
```

Decimal to Binary

```
if num < 0:
    is_negative = True
    num = abs(num)
else:
    is_negative = False

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

if is_negative:
    result = "-" + result
```

Decimal to Binary

```
if num < 0:
    is_negative = True
    num = abs(num)
else:
    is_negative = False

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

if is_negative:
    result = "-" + result
```

Decimal to Binary

```
if num < 0:
    is_negative = True
    num = abs(num)
else:
    is_negative = False

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

if is_negative:
    result = "-" + result
```

Decimal to Binary

```
if num < 0:
    is_negative = True
    num = abs(num)
else:
    is_negative = False

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

if is_negative:
    result = "-" + result
```

Decimal to Binary

```
if num < 0:
    is_negative = True
    num = abs(num)
else:
    is_negative = False

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

if is_negative:
    result = "-" + result
```

Decimal to Binary

```
if num < 0:
    is_negative = True
    num = abs(num)
else:
    is_negative = False

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

if is_negative:
    result = "-" + result
```

Definition of fractions in number systems

If $19_{10} = 10 * 10^1 + 9 * 10^0$, then what is 0.19_{10}?

[participation points for guesses]

Definition of fractions in number systems

The fraction $0.abc$ means

$$a * 10^{-1} + b * 10^{-2} + c * 10^{-3}$$

And for binary representation, we use the same idea $

$$a * 2^{-1} + b * 2^{-2} + c * 2^{-3}$$

Or in sentence form

The binary representation of a decimal fraction `f` would require finding the values of `a`, `b`, `c`, etc. such that

$$f = 0.5a + 0.25b + 0.125c + 0.0625d + ...$$

> Because $2^{-1} = 0.5$, and $2^{-2} = 0.25$, and $2^{-3} = 0.125$, etc.

Decimal fraction to binary fraction

Given that you know how to convert whole numbers to binary, how would you convert a

$3/8$ to a binary

$$frac38 = 0.375_{10} = 3 * 10^{-1} + 7 * 10^{-2} + 5 * 10^{-3}$$

A trick to convert to a decimal fraction to binary is to multiply it by a big enough power of 2 to turn it into a whole number

Has to be a power of 2 because we are converting to binary

Decimal fraction to binary fraction

Given

$$3/8 = 0.375_{10}$$

```
1. $0.375 * 2^1 = 0.75$
2. $0.375 * 2^2 = 1.5$
3. $0.375 * 2^3 = 3.0$
```

So we multiply by 2^3 to get a whole number

Then convert that whole number to binary

$$3_{10} = 11_2$$

Decimal fraction to binary fraction

Given

$$3/8 = 0.375_{10}$$

1. $0.375 * 2^1 = 0.75$
2. $0.375 * 2^2 = 1.5$
3. $0.375 * 2^3 = 3.0$

So we multiply by 2^3 to get a whole number

Then convert that whole number to binary

$$3_{10} = 11_2$$

Decimal fraction to binary fraction

Given

$$3/8 = 0.375_{10}$$

1. $0.375 * 2^1 = 0.75$
2. $0.375 * 2^2 = 1.5$
3. $0.375 * 2^3 = 3.0$

So we multiply by 2^3 to get a whole number

Then convert that whole number to binary

$$3_{10} = 11_2$$

Decimal fraction to binary fraction

Then divide it by 2^3 to get the binary fraction, shifting the bits to the right by 3 places

$$11_2/2^3 = 0.011_2$$

Because dividing by some power of 2 means shifting the bits to the right by that many places

Like how dividing by 10 means shifting the decimal point to the left by that many places

But

What if we *can't* find a power of 2 that turns it into a whole number?

This program relies on the fact that multiplying by 2 will eventually turn the fraction into a whole number

```python
x = 0.625

p = 0
while ((2**p) * x) % 1 != 0:
    print("Remainder =", (2**p) * x % 1)
    p += 1
num = int(x* (2**p)) # is now a whole number

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(result)
```

```python
x = 0.625

p = 0
while ((2**p) * x) % 1 != 0:
    print("Remainder =", (2**p) * x % 1)
    p += 1
num = int(x* (2**p)) # is now a whole number

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(result)
```

```python
x = 0.625

p = 0
while ((2**p) * x) % 1 != 0:
    print("Remainder =", (2**p) * x % 1)
    p += 1
num = int(x* (2**p)) # is now a whole number

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(result)
```

```python
x = 0.625

p = 0
while ((2**p) * x) % 1 != 0:
    print("Remainder =", (2**p) * x % 1)
    p += 1
num = int(x* (2**p)) # is now a whole number

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(result)
```

```python
x = 0.625

p = 0
while ((2**p) * x) % 1 != 0:
    print("Remainder =", (2**p) * x % 1)
    p += 1
num = int(x* (2**p)) # is now a whole number

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(result)
```

```python
x = 0.625

p = 0
while ((2**p) * x) % 1 != 0:
    print("Remainder =", (2**p) * x % 1)
    p += 1
num = int(x* (2**p)) # is now a whole number

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(result)
```

```python
x = 0.625

p = 0
while ((2**p) * x) % 1 != 0:
    print("Remainder =", (2**p) * x % 1)
    p += 1
num = int(x* (2**p)) # is now a whole number

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(result)
```

Issues

What is the output of

- `0.375`
- `0.5`
- `0.1`

Suppose that we want to represent

- $1 * 2^-3$, that would be 0.001_2 because it’s a power of 2
- but 0.1 is not a power of 2, so it can’t be represented exactly in binary
- so the output is long, potentially infinite, and has to be cut off at some point

Computers

Everything in computers is bits, because of that some numbers are just too big to fit in the bits available

So how are these bits actually computed in computer memory?

2 values, where one is the number and the other is the exponent

- [1, 1] -> $1 * 2^1 = 10_2 = 2.0_10$
- [1, -1] -> $1 * 2^{-1} = 0.1_2 = 1_10$
- [125, -2] -> 125.2^{-2} = 11111.01_2 = 31.25_10

Floating decimal point

We use a finite set of bits to represent a potentially infinite set of numbers

- the max number of digits governs the precision with which numbers can be represented
- Most modern computers use 32 bits to represent significant digits
- And if a number is too big, it is rounded to the nearest representable number, and the error will be at the 32nd bit
- the error only shows up on $2^{-32} \approx 2.3 * 10^{-10}$, which is very small

But even really small errors lead to large problems

```python
x = 0
for i in range(10):
    x += 0.125
print(x == 1.25) # true
```

```python
x = 0
for i in range(10):
    x += 0.1
print(x == 1) # false
```

So

- Never use `==` to compare floating point numbers
- Instead, check if the absolute difference is smaller than some small number, like 10^{-9} otherwise known as epsilon
- we need to be really careful when making applications that uses floating point numbers

We can make a better guess and check

Through approximation, when we want to find an answer where the answer isn’t an integer

Because

- Exact answers may not exist
- "good enough" answers may be sufficient

Finding the square root

- last time we made a guess and check algorithm to find the square root of a number
- but it only worked for perfect squares
- what if we wanted to find the square root of any positive integer

Question

- what does it mean to find the square root of x?

Usually it's

- find r such that $r * r = x$
- if x is not a perfect square, then r is not an integer

Approximation

1. find an answer that's good enough
    - like finding r such that $r * r$ is only a small distance away from x
    - epsilon ϵ is usually the term used for "small distance away"
    - where we want to find r such that $|r^2 - x| < \epsilon$
2. Algorithm
    - start with a guess, g, that we know is too small (like 0)
    - increment by a small value, `a`, to give a new guess `g`
    - check if `g**2` is close enough to `x` (within epsilon)
    - continue until we get an answer that's close enough
3. looking at all possible values `g + k*a` for integer values of `k`
4. stop when `|g**2 - x| < epsilon`

Line diagram here

Approximation

1. so we have two parameters to set
   - epsilon, how close are we to the answer
   - increment, how much to increase our guess by
2. And performance will vary based on these values
   - speed
   - accuracy

- lower increment means more steps, a slower program, but higher accuracy
- higher increment means fewer steps, a faster program, but lower accuracy, and may skip over the answer
- lower epsilon means more steps, a slower program, but higher accuracy
- higher epsilon means fewer steps, a faster program, but lower accuracy

Implementation

```
x = 36
epsilon = 0.01
num_guesses = 0
guess = 0.0
increment = 0.0001

while (abs(guess**2 - x) -x) >= epsilon:
    guess += increment
    num_guesses += 1

print("num_guesses =", num_guesses)
print(guess, "is close to square root of", x)
```

Implementation

```
x = 36
epsilon = 0.01
num_guesses = 0
guess = 0.0
increment = 0.0001

while (abs(guess**2 - x) -x) >= epsilon:
    guess += increment
    num_guesses += 1

print("num_guesses =", num_guesses)
print(guess, "is close to square root of", x)
```

Implementation

```
x = 36
epsilon = 0.01
num_guesses = 0
guess = 0.0
increment = 0.0001

while (abs(guess**2 - x) -x) >= epsilon:
    guess += increment
    num_guesses += 1

print("num_guesses =", num_guesses)
print(guess, "is close to square root of", x)
```

Implementation

```python
x = 36
epsilon = 0.01
num_guesses = 0
guess = 0.0
increment = 0.0001

while (abs(guess**2 - x) -x) >= epsilon:
    guess += increment
    num_guesses += 1

print("num_guesses =", num_guesses)
print(guess, "is close to square root of", x)
```

Implementation

```
x = 36
epsilon = 0.01
num_guesses = 0
guess = 0.0
increment = 0.0001

while (abs(guess**2 - x) -x) >= epsilon:
    guess += increment
    num_guesses += 1

print("num_guesses =", num_guesses)
print(guess, "is close to square root of", x)
```

Implementation

```
x = 36
epsilon = 0.01
num_guesses = 0
guess = 0.0
increment = 0.0001

while (abs(guess**2 - x) -x) >= epsilon:
    guess += increment
    num_guesses += 1

print("num_guesses =", num_guesses)
print(guess, "is close to square root of", x)
```

Implementation

```
x = 36
epsilon = 0.01
num_guesses = 0
guess = 0.0
increment = 0.0001

while (abs(guess**2 - x) -x) >= epsilon:
    guess += increment
    num_guesses += 1

print("num_guesses =", num_guesses)
print(guess, "is close to square root of", x)
```

Implementation

```
x = 36
epsilon = 0.01
num_guesses = 0
guess = 0.0
increment = 0.0001

while (abs(guess**2 - x) -x) >= epsilon:
    guess += increment
    num_guesses += 1

print("num_guesses =", num_guesses)
print(guess, "is close to square root of", x)
```

Implementation

Let’s try finding the cube root of

- 36
- 24
- 2
- 12345
- 54321

Will this loop always terminate?

Implementation

Let's try to debug

```python
x = 36
epsilon = 0.01
num_guesses = 0
guess = 0.0
increment = 0.0001

while (abs(guess**2 - x) -x) >= epsilon:
    guess += increment
    num_guesses += 1

    if num_guesses % 100000 == 0:
        print("current guess =", guess)
        print("current guess^2 =", guess**2)
        print("distance from x =", abs(guess**2 - x))
    if num_guesses % 1000000 == 0:
        input("continue?")

print("num_guesses =", num_guesses)
print(guess, "is close to square root of", x)
```

Implementation

Let's try to debug

```
x = 36
epsilon = 0.01
num_guesses = 0
guess = 0.0
increment = 0.0001

while (abs(guess**2 - x) -x) >= epsilon:
    guess += increment
    num_guesses += 1

    if num_guesses % 100000 == 0:
        print("current guess =", guess)
        print("current guess^2 =", guess**2)
        print("distance from x =", abs(guess**2 - x))
    if num_guesses % 1000000 == 0:
        input("continue?")

print("num_guesses =", num_guesses)
print(guess, "is close to square root of", x)
```

Implementation

Let’s try to debug

```
x = 36
epsilon = 0.01
num_guesses = 0
guess = 0.0
increment = 0.0001

while (abs(guess**2 - x) -x) >= epsilon:
    guess += increment
    num_guesses += 1

    if num_guesses % 100000 == 0:
        print("current guess =", guess)
        print("current guess^2 =", guess**2)
        print("distance from x =", abs(guess**2 - x))
    if num_guesses % 1000000 == 0:
        input("continue?")

print("num_guesses =", num_guesses)
print(guess, "is close to square root of", x)
```

We overshot

Diagram here, guess and guess ** 2

The fix

```
x = 36
epsilon = 0.01
num_guesses = 0
guess = 0.0
increment = 0.0001

while (abs(guess**2 - x) -x) >= epsilon and guess**2 <= x:
    guess += increment
    num_guesses += 1

print("num_guesses =", num_guesses)
if abs(guess**2 - x) >= epsilon:
    print("Failed to find the square root of", x)
else:
    print(guess, "is close to square root of", x)
```

The fix

```
x = 36
epsilon = 0.01
num_guesses = 0
guess = 0.0
increment = 0.0001

while (abs(guess**2 - x) -x) >= epsilon and guess**2 <= x:
    guess += increment
    num_guesses += 1

print("num_guesses =", num_guesses)
if abs(guess**2 - x) >= epsilon:
    print("Failed to find the square root of", x)
else:
    print(guess, "is close to square root of", x)
```

The fix

```python
x = 36
epsilon = 0.01
num_guesses = 0
guess = 0.0
increment = 0.0001

while (abs(guess**2 - x) -x) >= epsilon and guess**2 <= x:
    guess += increment
    num_guesses += 1

print("num_guesses =", num_guesses)
if abs(guess**2 - x) >= epsilon:
    print("Failed to find the square root of", x)
else:
    print(guess, "is close to square root of", x)
```

The fix

```
x = 36
epsilon = 0.01
num_guesses = 0
guess = 0.0
increment = 0.0001

while (abs(guess**2 - x) -x) >= epsilon and guess**2 <= x:
    guess += increment
    num_guesses += 1

print("num_guesses =", num_guesses)
if abs(guess**2 - x) >= epsilon:
    print("Failed to find the square root of", x)
else:
    print(guess, "is close to square root of", x)
```

Now it stops if it overshoots and reports an error

But what if we don’t want to fail? What can we do

[participation points for guesses and ideas]

Now it stops if it overshoots and reports an error

But what if we don’t want to fail? What can we do

[participation points for guesses and ideas]

hint: think of the values that we set in the very beginning

Remember

Overshooting can happen

- Always set another end condition

Be careful when comparing floating point numbers

Recap

- can't use `==` to check an exit condition
- need to be careful that looping mechanisms don't jump over the exit condition
- tradeoffs exist between efficiency and accuracy
- need to think about how close an answer we want when setting parameters of an algorithm
- to get a good answer, this method can be slow

We'll figure out how to speed it up next time