Loops and Iteration

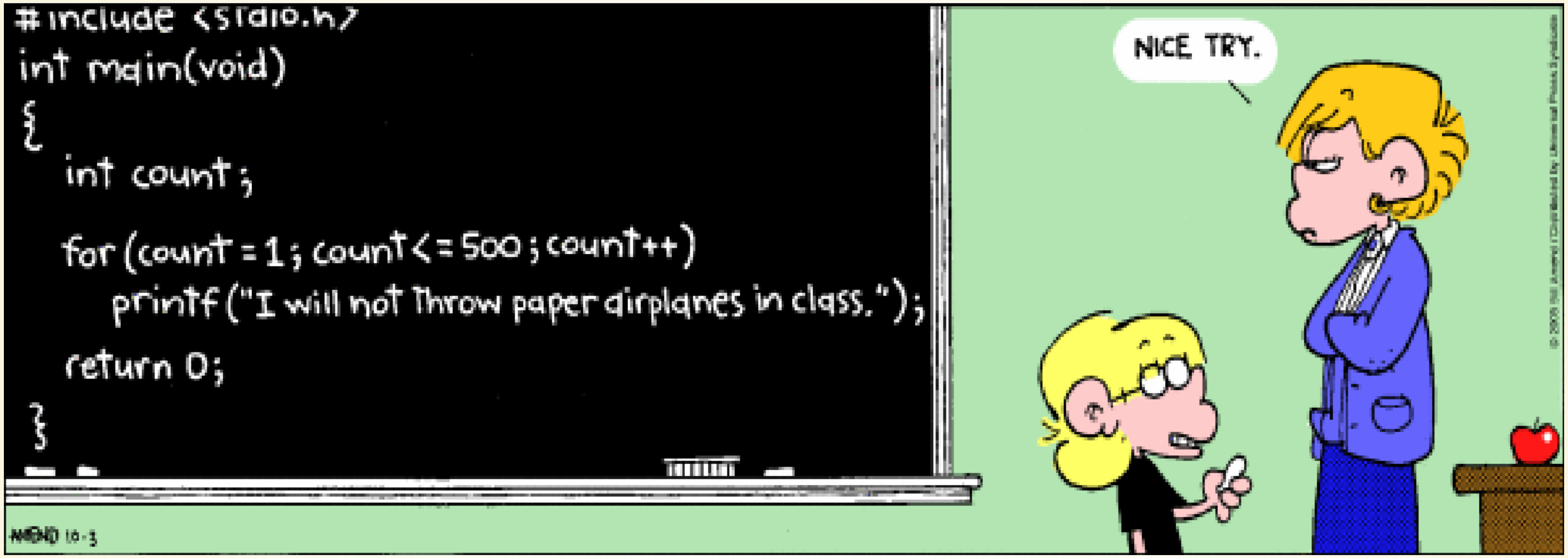
adapted from material by Mike Scott and Bill
Young at the University of Texas at Austin

Repetitive Activity

Sometimes, we need to do the same thing many times.

So let's be a little clever about it!

Loops allow us to repeat things multiple times.



Note: we don't actually have to do the *exact* same thing over and over---we can change it a little bit.

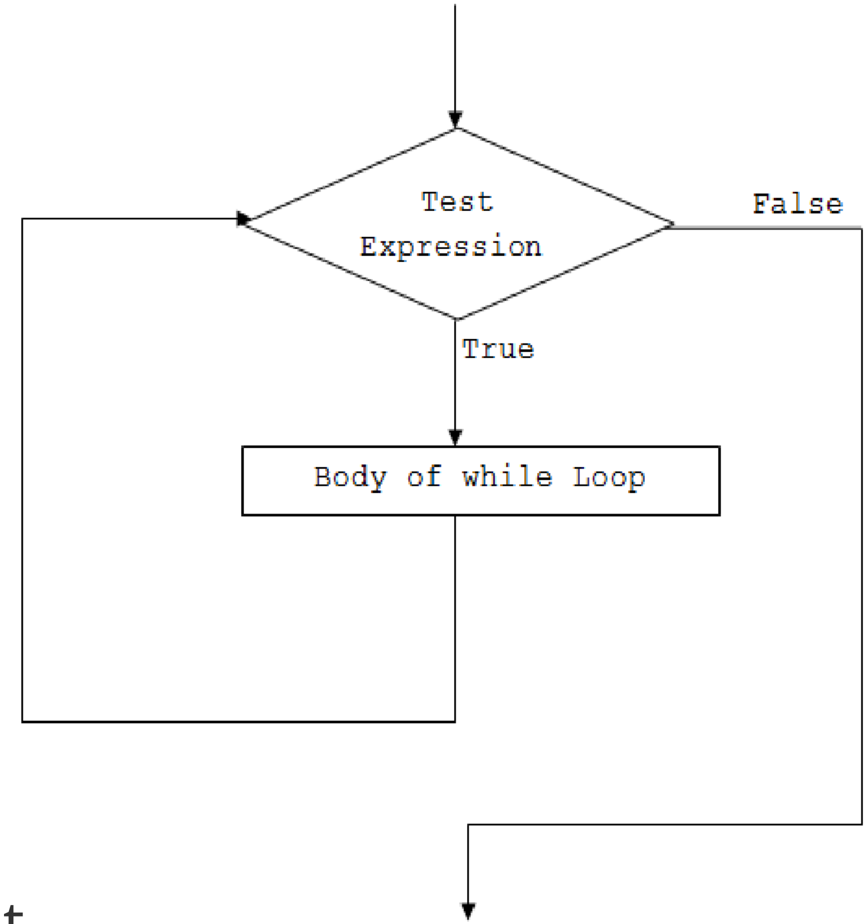
Computers can do billions of operations a second. Loops are how we harness this power!

```python
def main():
  text = input("Please enter the number three: ")
  value = int(text)
  num_times_failed = 0
  while value != 3:
    text = input("That was not three! Please enter the number three: ")
    value = int(text)
    num_times_failed += 1

  if num_times_failed < 4:
        print("Thank you!")
  else:
    print("Took you long enough")

main()
```

While Loop

The **while loop** lets us repeat operations. General form:

```
while condition:
    statement_1
    statement_2
    ...
```



As long as the condition is true, the loop continues to run.

All statements in the loop body must be indented the same amount.

```
1 # Print out our punishment lines
2 def main():
3   count = 500
4   message = "I will not throw paper airplanes in class."
5   i = 0
6   while i < count:
7     print(i, message)
8     i += 1
9
10 main()
```

What if we forgot line 8?

Warm Up: Printing Squares

> *Take a number N from the user. Print all perfect squares that are less than or equal to N.*

A number N is a perfect square if there is an integer Z such that $Z^2 = N$

9, 16, 25, 36, etc.

Primality Testing

An integer is prime if it is greater than 1 and has no integer divisors except 1 and itself.

To test whether an integer n is prime, see if any number in $[2, 3, ..., n-1]$ divides it with no remainder.

You cannot do this without loops without knowing n in advance. **Why not?**

Primality Testing

Write a program which takes a number from the user and decides whether it is prime or not.

An integer is prime if it is greater than 1 and has no integer divisors except 1 and itself.

To test whether an integer n is prime, see if any number in $[2, 3, \ldots, n-1]$ divides it with no remainder.

```python
def main():
    x = int(input("Enter a number: "))

    is_prime = True
    divisor = 2
    while divisor < x:
        if x % divisor == 0:
            is_prime = False
        divisor += 1

    if is_prime:
        print(x, "is prime.")
    else:
        print(x, "is not prime.")

main()
```

Timing

Input	Time (Desktop)
37	11.94 ms
176970203	7.03 s
479001599	18.91 s
479001600	19.01 s

This code **works**, but it's not **fast**.

Let's go faster!

How can we speed this program up?

- We don't need to check all multiples of two! **Why?**
- We don't need to go up to $n - 1$. What's the largest number we need to go up to?
- What if we discover that the first factor divides the number? Do we need to keep checking?

```python
import math
def main():
    x = int(input("Enter a number: "))

    is_prime = x % 2 != 0
    divisor = 3
    limit = math.sqrt(x)
    while divisor < limit and is_prime:
        if x % divisor == 0:
            is_prime = False
        divisor += 2

    if is_prime:
        print(x, "is prime.")
    else:
        print(x, "is not prime.")

main()
```

Input	Old Time	New Time	Speedup
37	11.94 ms	11.87 ms	1x
176970203	7.03 s	11.92 ms	**587x**
479001599	18.91 s	13.05 ms	**1449x**
479001600	19.01 s	11.91 ms	**1596x**

The new times suggest that the main contributor to timing is *printing the result*.

Previously, with straight-line code, how long the program took was basically limited by how much code we could write. Now, with loops, we can make programs that take a very long time.

Computer scientist spend a *lot* of time trying to improve the efficiency of algorithms.

With the right languages and algorithms, you can get *very* fast!

```cpp
int main() {
  int64_t num = 3318308475676071413;
  std::cin >> num;
  bool isPrime = true;
  if (num <= 2 || num % 2 == 0 || num % 3 == 0 || num % 5 == 0) {
    isPrime = false;
  }
  int wheel[8] = {7, 11, 13, 17, 19, 23, 29, 31};
  for (int i = 0; i < sqrt(num); i += 30) {
    for (int c : wheel) {
      if (c > sqrt(num))
        break;
      if (num % (c + i) == 0) {
        isPrime = false;
        break;
      }
    }
    if (!isPrime)
      break;
  }
  if (isPrime)
    std::cout << num << " is prime" << std::endl;
  else
    std::cout << num << " is not prime" << std::endl;
}
```

Timings

Test primality of 3318308475676071413

Python, version 1: Way too long

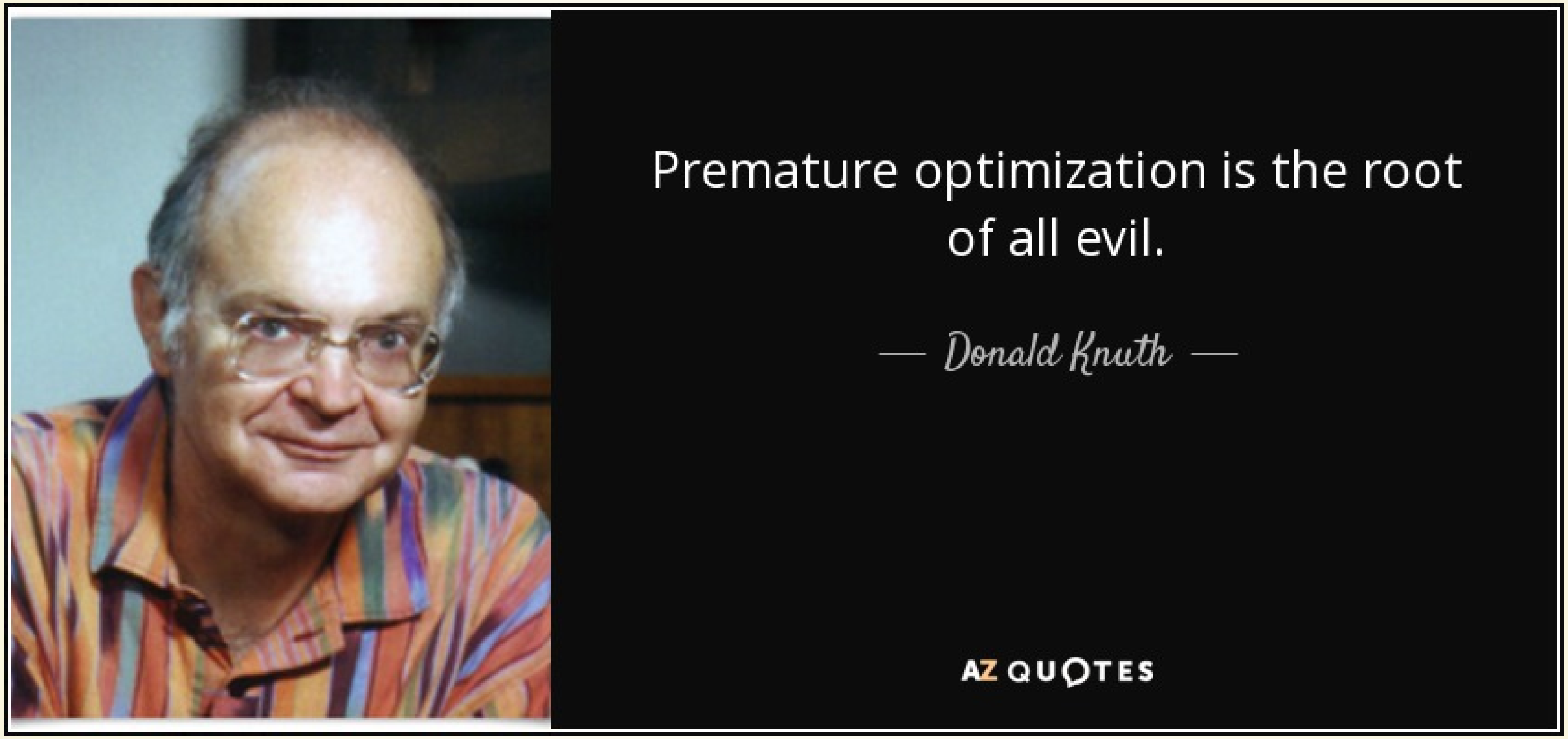
Python, version 2: **39.65 seconds**

C++ w/ wheel factorization: **699 ms**

By using the right tricks on the C++ version, I could probably get another 8x-10x speedup.

Total speedup over Python version 1: **over 9,000,000x**!!

A Word of Warning



In This Class

As long as your code runs in reasonable time (under 1 minute) for the things it needs to do, I don't really care about speed.

In General

Think carefully about why you need the program to be fast, and **measure** it to figure out what needs to be sped up.

Square Roots

Warm-up

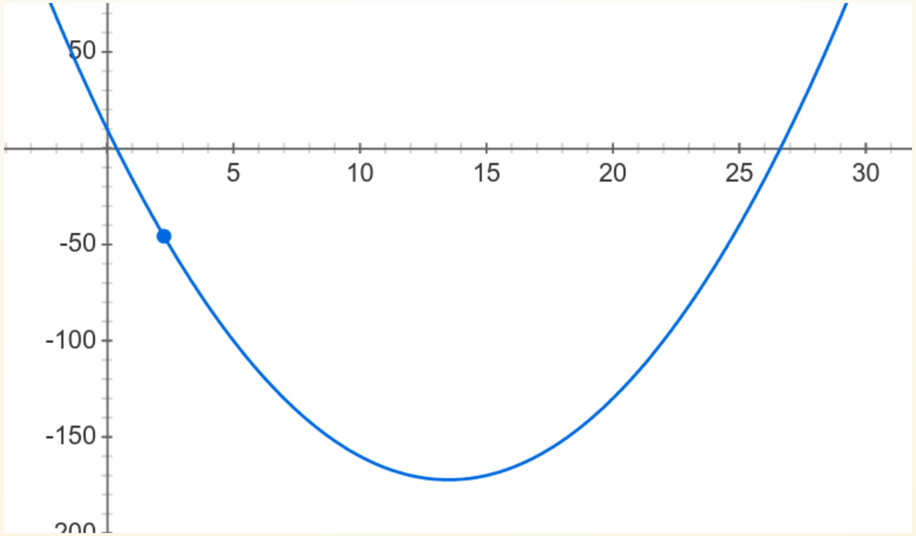
```python
# Count even numbers from bot to top
top = int(input("Enter a top number:"))
bot = int(input("Enter a bottom number:"))

x = 0
while x <= top:
  if x % 2 == 0:
    print(x)
    x += 1
  elif x == top:
    print(x)
    print("And we're done!")
  else:
    x += 1
```

Warmup: Calculate Approximate Minimum

Consider the following function:

$$f(x) = x^2 - 27 * x + 10$$

Calculate the approximate minimum of this function by stepping with a while-loop.



Suppose I give you a number x. I want you to find a number y such that $y^2 = x$, or $y = \sqrt{x}$. How do you do this?

Worst Idea Ever

```python
import random
import math

# Approximate the square root of a positive
# integer by random guessing
def main():
  x = int(input("Enter a number: "))

  while True:
    y = random.rand() * x
    if abs(y ** 2 - x) < 0.1:
      print(y, "is approximately the square root of", x)
      break

main()
```

We have no idea how long this will take!

Slightly Better Idea

```python
import math

# Approximate the square root of an integer very slowly
def main():
  num = int(input("Enter a positive integer: "))
  while num < 0:
    print("That wasn't positive.")
    num = int(input("Enter a positive integer: "))
  guess = 0.0
  while guess ** 2 < num:
    guess += 0.01
  print("The square root of", num, "is about", guess)

main()
```

```
~/tmp via 🐍 v3.10.4
18:01:54 ∈ ❯ python sqrt.py
Enter a positive integer: 3
The square root of 3 is about 1.7400000000000013
~/tmp via 🐍 v3.10.4
18:01:56 ∈ ❯ python sqrt.py
Enter a positive integer: 10
The square root of 10 is about 3.1699999999999764
~/tmp via 🐍 v3.10.4
18:01:58 ∈ ❯ python sqrt.py
Enter a positive integer: -10
That wasn't positive.
Enter a positive integer: 100
The square root of 100 is about 10.009999999999831
```

Note that the last guess isn't accurate! Foiled again by the approximate nature of floating-point arithmetic.

How would you change the code to get a better approximation?

Another Idea

feat: calculus!

Consider the following function:

$$f(x) = \sqrt{x} - 10$$

When is it zero?

How can we find the zero?

$$f(x) = \sqrt{x} - 10$$

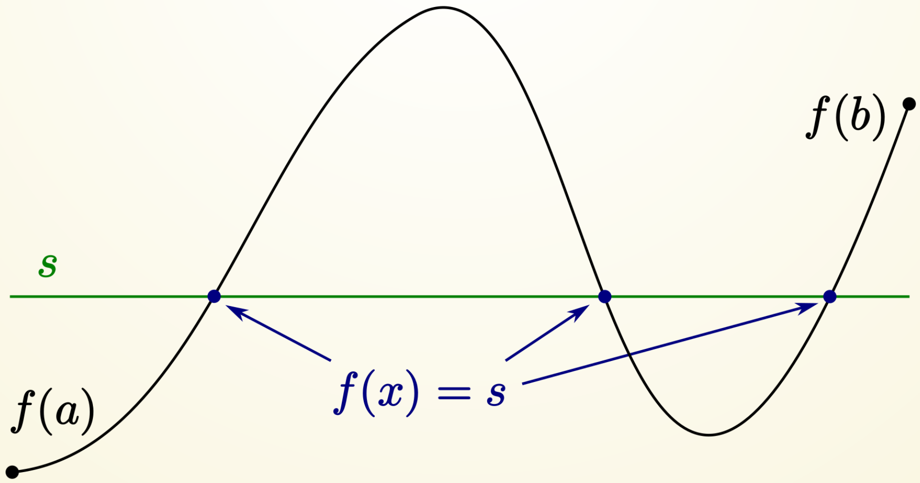
$$f(0) = -10$$

$$f(26) = 4$$

What do we know about f between these two x-values?

Intermediate Value Theorem

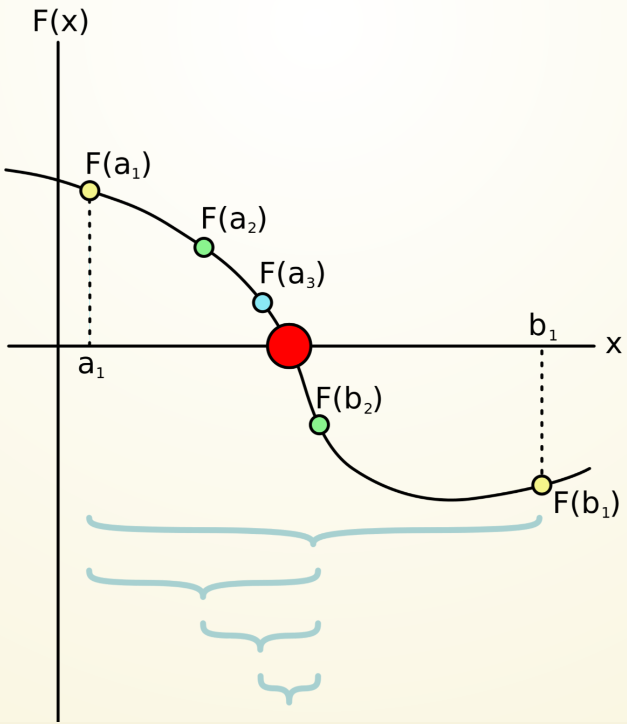
If f is a continuous function on $[a, b]$, then for any $s \in [f(a), f(b)]$, there exists $x \in [a, b]$ such that $f(x) = s$.



Root Finding: Bisection

Find a such that $f(a) < 0$ and b such that $f(b) > 0$.

Then, continually narrow the interval so that the condition remains true.



Write a program to find the square root of a number by bisection.

For Loops

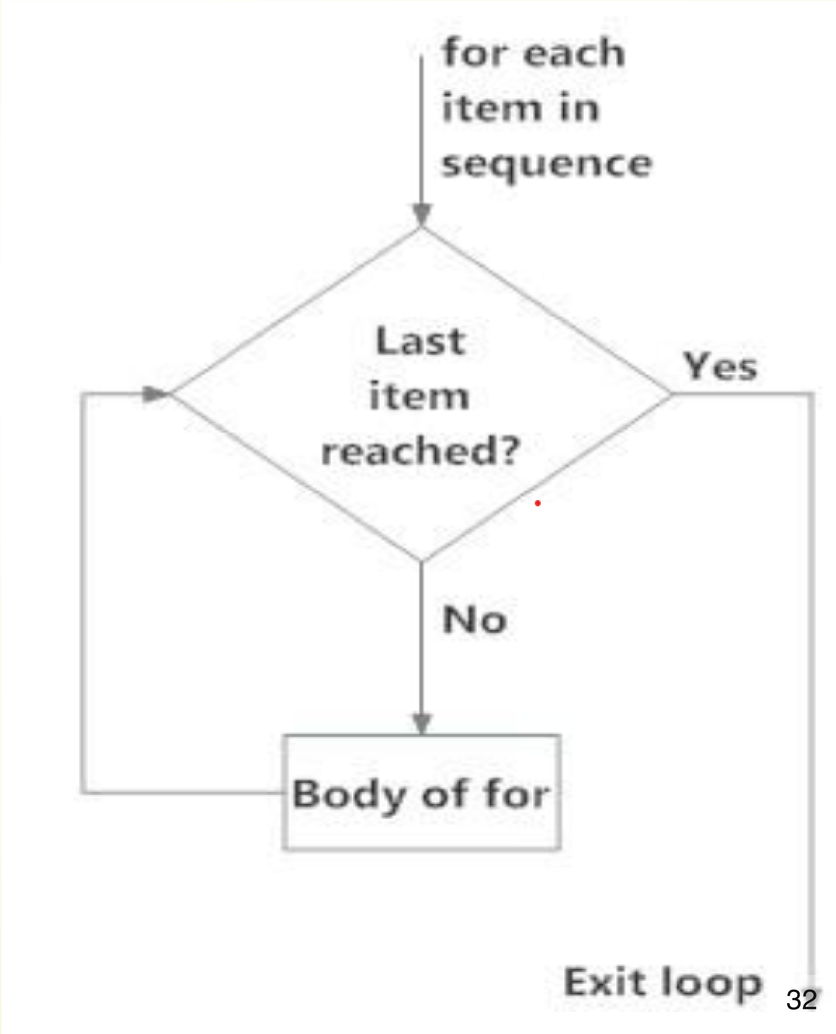
In a for-loop, you usually know how many times you'll execute.

General form:

```
for var in sequence:
    statement_1
    statement_2
```

Meaning: for each element in sequence, assign var to the element and then execute the statements.

Note: indentation must be the same for all body stmts.



What's a Sequence?

A "sequence" is a general term for anything with multiple items stored one after another.

```
# a list is a sequence
seq = [2, 3, 5, 7, 11, 13]
```

The range() function is a good way to generate a sequence.

range(a,b): generates the sequence $[a, a+1, \ldots, b-1]$

range(b): is the same as range(0, b)

range(a,b,c): generates $[a, a+c, a+2c, \ldots, b']$ where b' is the last value that is less than b.

```
>>> for i in range(3, 6): print(i, end="")
...
345>>> for i in range(3, 6): print(i, end=" ")
...
3 4 5 >>> for i in range(3): print(i, end=" ")
...
0 1 2 >>> for i in range(0, 11, 3): print(i, end=" ")
...
0 3 6 9 >>> for i in range(11, 0, -3): print(i, end=" ")
...
11 8 5 2 >>>
```

Let's write a program to print a table of the power of a given base up to N.

e.g. $7^1, 7^2, \ldots, 7^N$

Warmup

Write a program which computes the sum of the numbers from 1 to N. Do this with a for-loop.

(No need to use input() for this---just put an `N = 10` or something at the top of your file).

While vs For

Nested Loops

The body of a loop can contain any kind of statement, **including other loops.**

Let's write a program to print out BMI values for heights between 54 and 82 inches (going up by 2 inches each time), and weights from 85 to 350 pounds (going up by 5 pounds).

It is arbitrary which loop the outer loop is.

break and continue

Sometimes we don't always want to wait until the end of a loop to do something

break

break lets us exit a loop early

```python
x = 0
while x < 10:
    x += 1
    if x == 7:
        break
    else:
        print("x is", x)
```

continue

continue lets us skip an iteration of the loop. Instead of exiting, we immediately go to the top of the loop when we execute a continue

```python
x = 0
while x < 10:
    x += 1
    if x % 2 == 1:
        continue
    print("x is", x)
```

In theory, you don't need **break** and **continue** to write programs in Python!

In practice, it makes certain tasks a *lot* nicer.

Silly Encryption

Hide the true message inside a string by putting in lots of 'q's and '2's.

If you see a '7', the message stops there (everything else is designed to fool you).

" qqh2eql22lqqo2q2 q2d2qqqa22q22rq22q2kq2qn2q222eq2q2s2q2q2s2q2q2 2q2q2qm2qyqqq qqoqlqqdqqqqq f2222rq2qqqi22qeqqqq2nqqq222d7i2aavea2a22222 a2q2q2q2q2q2 qqqs22222eqq22qqcqqq22qqrqq22qq2q2eqqqq2q22qt

f-strings

Mixing data with strings

So far, when we wanted to print data, we used the feature of print that lets us print multiple things:

```
apples = int(input("How many apples"))
print("You have", apples, "apples.")
```

This works well enough, but sometimes we'd like to have finer control over what we're printing.

```
place = int(input("What place did the racer finish?"))
print("The racer finished in", place, "th place.")
```

Result:

> *The racer finished in 17 th place.*

We want:

> *The racer finished in 17th place.*

Enter f-strings

```python
place = int(input("What place did the racer finish?"))
print(f"The racer finished in {place}th place.")
```

You place an f at the front of the string (before the opening quotation marks).

Within the curly braces ({}) goes a Python expression to evaluate. This can be python code!

```python
num = int(input("Enter a number:"))
print(f"Twice {num} is {2 * num}")
```

You do not have to use them, but f-strings make many things easier to print.

```python
name = input("What is your name?")
print(f"{name.upper()} IS AWESOME!")
```

Just don't forget the f at the front!

```python
print("The result is {3 * 7}")
```

```
>>> print("The result is {3 * 7}")
The result is {3 * 7}
```

Practice!

Blastoff

Print a countdown from 20 to 1,
then print "BLASTOFF".

Make this program as short and
simple as possible.

Factorial

Use a for-loop to compute the factorial of a number.

Harmonic Series

Print the first N partial sums of the harmonic series.

In mathematics, the **harmonic series** is the infinite series formed by summing all positive unit fractions:

$$\sum_{n=1}^{\infty} \frac{1}{n} = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \cdots .$$

Retirement

Suppose we invest $6000 a year into a retirement account. How much does this money grow over 30 years, assuming we have various rates of return between 1% and 9%?

Coin Flipping

How many fair coin tosses do we need to see 10 of the same side (either H or T) in a row?

Repeat this experiment many times (e.g. 2500) and average over all the results.