CS303E: Elements of Computers and Programming

Repitition with Loops

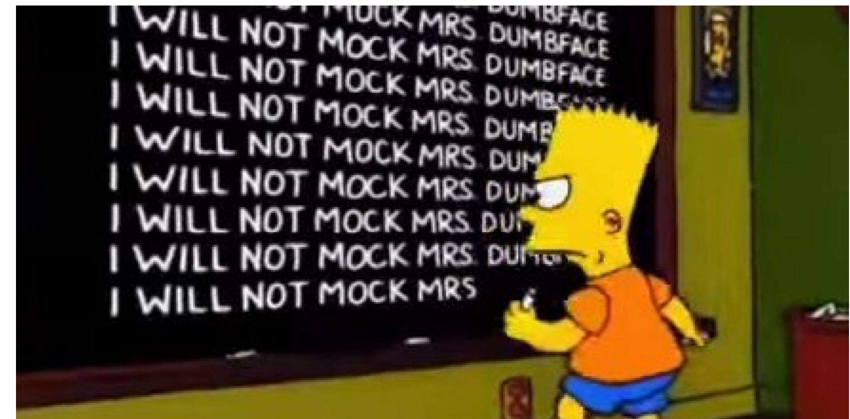
Mike Scott
Department of Computer Science
University of Texas at Austin

Adapted from
Professor Bill Young's Slides

Last updated: May 30, 2024

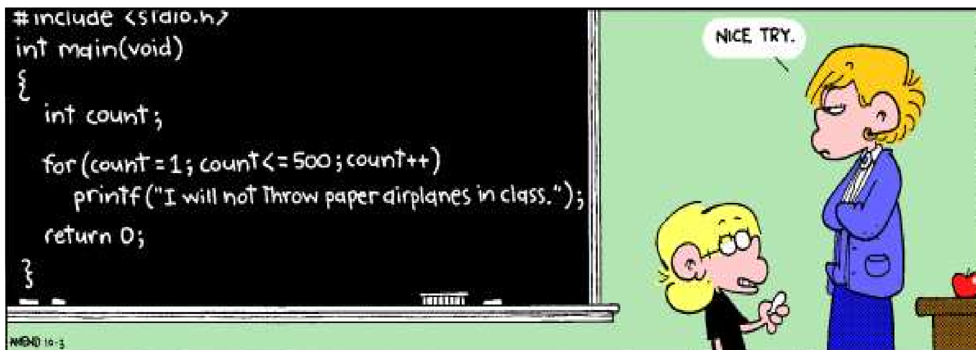
Repetitive Activity

Often we need to do some (program) activity numerous times:



Using Loops

So we might as well use cleverness to do it.
That's what loops are for.



It doesn't have to be the exact same thing over and over.

And this is how we really harness the power of a computer that can perform tens of billions (or more) computations per second!

While Loop

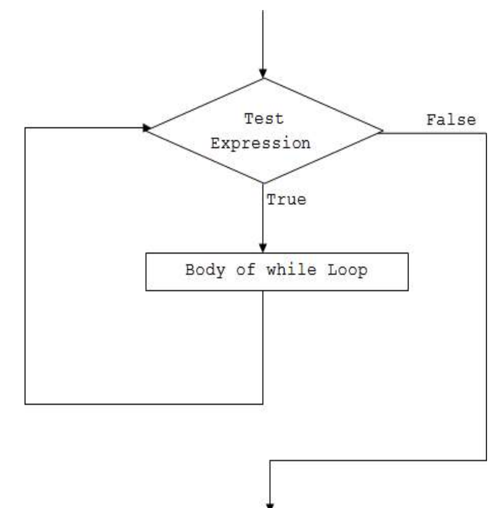
The majority of programming languages include syntax to **repeat** operations.

while loop is one option. General form:

```
while condition:
    statement(s)
```

Meaning: as long as the condition is true when checked, execute the statements.

As with conditionals (if/elif/else), all of the statements in the body of the loop must be indented the same amount.



While Loop

In file not_throw_airplanes.py:

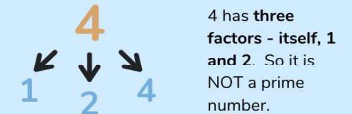
```
# Print out I will not throw paper airplanes in class
# 500 times.
def main():
    COUNT = 500
    MESSAGE = "I will not throw paper airplanes in class."
    i = 0
    while i < COUNT:
        print(i, MESSAGE)
        i += 1


main()
```

What would happen if we forgot the i += 1?

```
0 I will not throw paper airplanes in class.
1 I will not throw paper airplanes in class.
2 I will not throw paper airplanes in class.
3 I will not throw paper airplanes in class.
4 I will not throw paper airplanes in class.
```


While Loop Example: Test Primality

How do prime numbers work?



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

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

You couldn't do that in *straight line* code without knowing n in advance. Why not?

Even then it would be *really* tedious if n is very large.

is_prime_1 Loop Example

is_prime_1.py

```
def main():
    number = int(input("Please enter a number greater than"
                       + " or equal to 2: "))
    prime = True
    divisor = 2
    while divisor < number and prime:
        prime = number % divisor != 0
        divisor += 1
    if prime:
        print(number, "is prime.")
    else:
        print(number, "is not prime.")
    # OR print(number, " is",
    #      "not" if not prime else "", " prime", sep="")

main()
```


is_prime_1 Loop

```
Please enter a number greater than or equal to 2: 37
37 is prime.
```

```
Please enter a number greater than or equal to 2: 176970203
176970203 is prime.
```

The second example took ~24 seconds to complete on my laptop.

It works, though it's pretty inefficient. If a number is prime, we test every possible divisor in [2 ... n-1].

- If n is not prime, it will have a divisor less than or equal to $\sqrt{n}$.
- There's no need to test any even divisor except 2.

A Better Version: is_prime_2.py

```python
import math


def main():
    """Determine if a number entered by the user is prime or not."""
    number = int(input("Please enter a number greater than"
                       + " or equal to 2: "))

    # Special case for 2, the only even prime.
    prime = number == 2 or number % 2 != 0
    # If number is not even then we only need to divide
    # by odd numbers.
    divisor = 3

    limit = math.sqrt(number)
    while divisor <= limit and prime:
        prime = number % divisor != 0
        divisor += 1
    if prime:
        print(number, "is prime.")
    else:
        print(number, "is not prime.")
    # OR print(number, " is",
    #       "not" if not prime else "", " prime", sep="")


main()
```

The Better is_prime_2 Version

is_prime_1 does 176,970,202 divisions to discover that 176_970_203 is prime.

is_prime_2 does "only" 13,302.

Took much less than a second to complete.

Computer scientists and software developers spend a lot of time trying to improve the efficiency of their programs and algorithms.

Measurably reduce the number of computations.

Example While Loop: Approximate Square Root

You could approximate the square root of a positive integer as follows: square_root.py

```python
# Approximate the square root of a positive
# integer VERY SLOWLY by increments of 0.1
def main():
    number = int(input("Enter a positive integer: "))
    while number < 0:
        print(number, 'isn\'t a positive int')
        number = int(input("Enter a positive integer: "))
    guess = 0.1
    while guess ** 2 < number:
        guess += 0.1
    print('The square root of', number,
          'is approximately equal to ', guess)


main()
```


Running the Example

```
Enter a positive integer: -37
-37 isn't a positive int
Enter a positive integer: -12
-12 isn't a positive int
Enter a positive integer: -891273
-891273 isn't a positive int
Enter a positive integer: 1_024_237
The square root of 1024237 is approximately equal to  1012.1000000001616
```

```
Enter a positive integer: 100
The square root of 100 is approximately equal to  10.09999999999998
```

Notice that the last one isn't quite right. The square root of 100 is exactly 10.0. Foiled again by the approximate nature of floating point numbers and floating point arithmetic.

More efficient way of calculating square root?

Newton's method for approximating square roots adapted from the Dr. Math website

The goal is to find the square root of a number. Let's call it num

1. Choose a rough approximation of the square root of num, call it approx.

    How to choose?

2. Divide num by approx and then average the quotient with approx,

    in other words we want to evaluate the

    expression ((num/approx) + approx) / 2

3. How close are we? In programming we would store the result of the expression back into the variable approx.

4. How do you know if you have the right answer?

For Loop

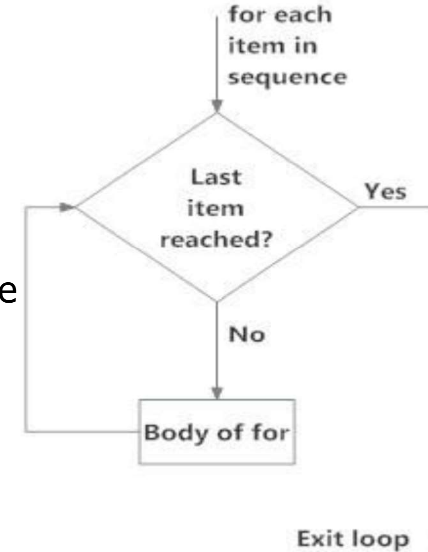
In a for loop, you typically know how many times you'll execute.

General form:

```
for <var> in <sequence>:
    <statement(s)>
```

Meaning: assign each element of sequence in turn to var and execute the statements.

As usual, all of the statements in the body must be indented the same amount.



What's a Sequence?

A Python sequence holds multiple items stored one after another.

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

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

range(a, b) : denotes the sequence a, a+1, ..., b-1.

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

range(a, b, c) : generates a, a+c, a+2c, ..., b', where b' is the last value < b.

Range Examples

```
>>> 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
>>>
```

For Loop Example

Suppose you want to print a table of the powers of a given base up to base^n. In file powers_of.py:

```
# Print the powers of a base entered by the user up to
# the nth power, also entered by the user.
def main():
    base = int(input('Enter the base: '))
    max_power = int(input('Enter the maximum power: '))
    for power in range(0, max_power + 1):
        print(base, 'to the', power, 'is',
              base ** power)


main()
```


For Loop Example

```
Enter the base: 2
Enter the maximum power: 42
2 to the 0 is 1
2 to the 1 is 2
2 to the 2 is 4
2 to the 3 is 8
2 to the 4 is 16
2 to the 5 is 32
2 to the 6 is 64
2 to the 7 is 128
2 to the 8 is 256
2 to the 9 is 512
2 to the 10 is 1024
2 to the 11 is 2048
2 to the 12 is 4096
2 to the 13 is 8192
2 to the 14 is 16384
2 to the 15 is 32768
2 to the 16 is 65536
2 to the 17 is 131072
2 to the 18 is 262144
2 to the 19 is 524288
```

```
Enter the base: 1037
Enter the maximum power: 12
1037 to the 0 is 1
1037 to the 1 is 1037
1037 to the 2 is 1075369
1037 to the 3 is 1115157653
1037 to the 4 is 1156418486161
1037 to the 5 is 1199205970148957
1037 to the 6 is 1243576591044468409
1037 to the 7 is 1289588924913113740133
1037 to the 8 is 1337303715134898948517921
1037 to the 9 is 1386783952594890209613084077
1037 to the 10 is 1438094958840901147368768187849
1037 to the 11 is 1491304472318014489821412610799413
1037 to the 12 is 154648273779378102594480487739899128
```


Nested Loops

The body of while loops and for loops contain any kind of statements, **including other loops.**

Suppose we want to compute and print out the BMI value for heights from 4' 6" (4 feet, 6 inches = 54 inches) to 6' 10" (82 inches) going up by 2 inches each time
AND weights from 85 to 350 pounds, going up by 5 pounds?

We could then take that data and create a visual graph for quick look up.

It is arbitrary whether the *outer loop* is height or weight

Print BMI for various heights and weights

```
# Print out BMI (Body Mass Index) values for heights from for
# heights from 4' 6" (4 feet, 6 inches = 54 inches)
# to 6' 10" (82 inches) going up by 2 inches each time
# AND weights from 85 to 350 pounds, going up by 5 pounds.
def main():
    english_units_conversion = 703
    for height in range(54, 83, 2):
        print('current height =', height)
        for weight in range(85, 351, 5):
            bmi = english_units_conversion * weight / (height ** 2)

            # Below is an example of the format function.
            # < means left justify
            # 4 means 4 total spots
            # .1 means 1 digit after the decimal
            # f means a floating point number
            print('height =', height, 'weight =', weight,
                  'bmi =', format(bmi, '<4.1f'))
```