Week 5

Debugging, Loops and Iteration

Learning Objectives

Basic Debugging

- Describe a proper debugging process
- Describe the use of an interactive debugger tool in a programming IDE

Loops and Iteration

- Identify a repetition process from an ordering of steps or a flow chart
- Create a while loop in Python, and correctly form the conditional statement
- Recognize an infinite loop, and how to correct one
- Create a for loop in Python, and correctly define a range for iteration
- Identify values of variables and iterators for each iteration through a loop
- Use nested looping statements

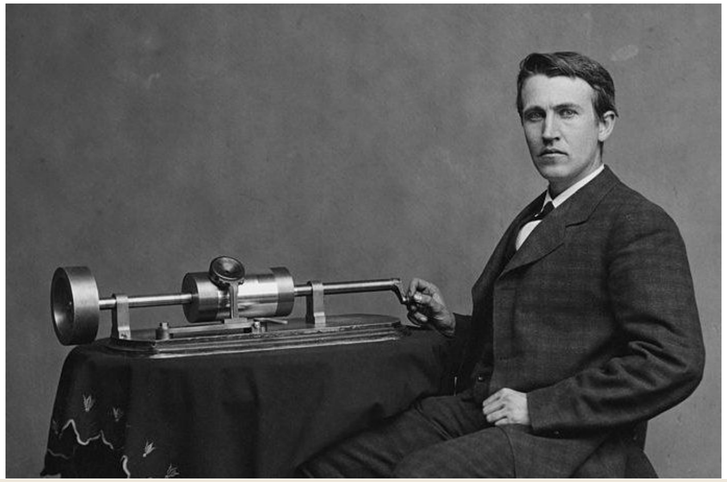
Simple Debugging

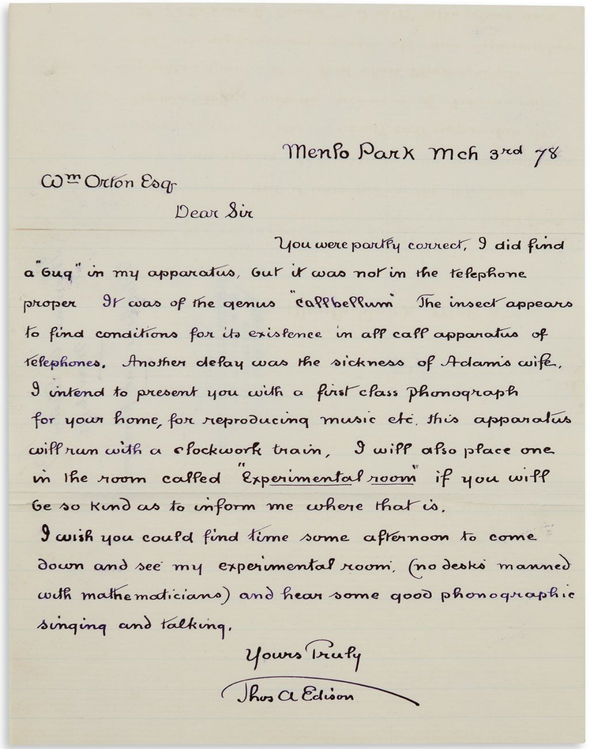
Bugs

Bug is the common term used in computing for "a problem in a program"

The term goes back to Thomas Edison

- Working on improving the telegraph, and fixed a problem with something he called a "bug trap"
- He continued using the term for problems he encountered


Thomas Edison and his early phonograph, circa 1877. LIBRARY OF CONGRESS/PUBLIC DOMAIN



Menlo Park Mch 3rd 78

Wm Orton Esqr

Dear Sir

You were partly correct, I did find a "bug" in my apparatus, but it was not in the telephone proper It was of the genus "callbellum" The insect appears to find conditions for its existence in all call apparatus of telephones. Another delay was the sickness of Adams wife. I intend to present you with a first class phonograph for your home, for reproducing music etc. this apparatus will run with a clockwork train, I will also place one in the room called "Experimental room" if you will be so kind as to inform me where that is.

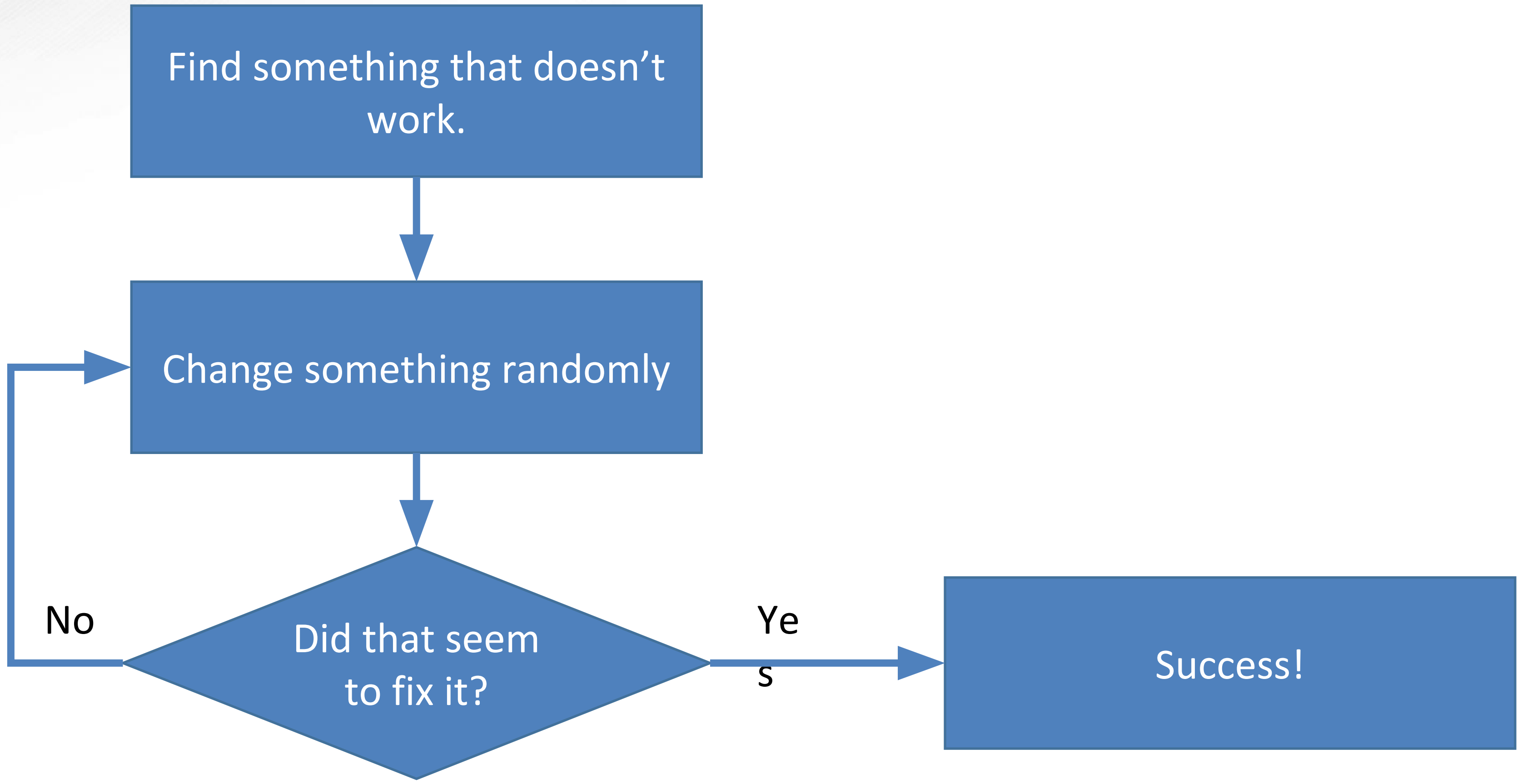
I wish you could find time some afternoon to come down and see my experimental room, (no desks manned with mathematicians) and hear some good phonographic singing and talking.

Yours Truly

Thos A Edison

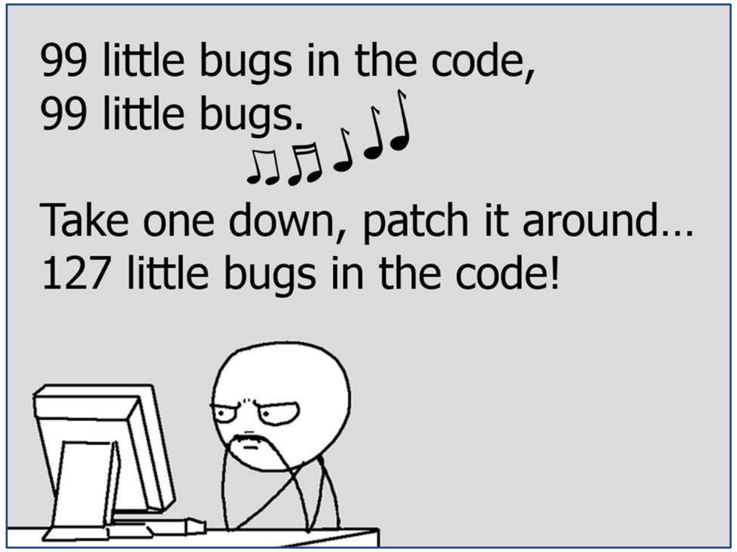
Thomas A. Edison's letter to Western Union President William Orton, 1878. COURTESY OF SWANN AUCTION GALLERIES

Debugging process – Inexperienced programmers



A better debugging process

1. Find a repeatable problem (e.g., A test case that reliably gives the wrong answer)
2. Isolate issue location in code (Run code to find where things seem to go wrong)
3. Examine code to find the specific bug
4. Try to fix the bug
5. Test whether it now passes the test case that was causing trouble
6. Check to make sure nothing else broke
7. Consider whether other places in code are likely to have a similar problem


99 little bugs in the code,
99 little bugs. ♫♬♪♩♩
Take one down, patch it around...
127 little bugs in the code!

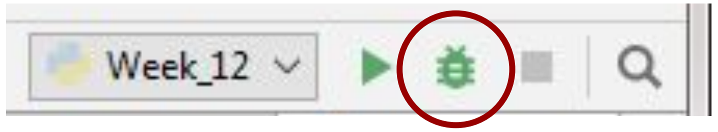
Interactive Debugger

- Many IDEs include a debugger (Including PyCharm)
- A debugger does not *fix* your bugs! It is a tool to help *you* debug your code.
- Debuggers usually let you do two main things:
  - Control the execution of the code
  - Examine the values stored in memory in the code
- We'll go over some of the most useful specific things that are found in most debuggers.
  - There are usually a lot of other tools, more advanced but less frequently needed

Running the debugger

You usually need to run in “debug mode” separately from the “normal” mode.

– In PyCharm, you click the green bug-looking icon



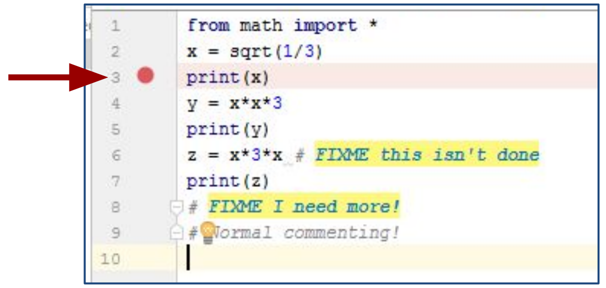
This will start the program

– Your code will not run as efficiently, since the executable is created for debugging purposes – extra information is provided, less optimization used.

Debugger – execution control tools available

Breakpoints

- Mark a line of code as a breakpoint. Usually set by clicking in front of the line of code (creates a red circle in PyCharm)
- Code will execute to that point and then pause



```
from math import *
x = sqrt(1/3)
print(x)
y = x*x*3
print(y)
z = x*3*x # FIXME this isn't done
print(z)
# FIXME I need more!
#Normal commenting!

```

Step-over

- Executes next line of code. If there is a function call, the function call is made and returns normally before going to the next line of code

Step-into

- Executes next line of code. If there is a function call, the code stops inside that function, at the beginning of the function

Run

- Runs until a breakpoint is encountered

Debugger – Examination Tools available

The Call Stack

- Lists the “stack” of functions that have been called. If function A calls B, and B calls C, then when in C, your stack will show C on top, then B, then A.

A Variable Watch-List

- A list of some (sometimes all) of the current variables, and their values
- You can usually add/remove variables from the watchlist

Using the Debugger

- Usually set a breakpoint just *before* code you suspect is problematic.
- Run to the breakpoint, then step-over until you see a variable with a wrong value.
  - You might see the error as you go through line-by-line, but not always.
- When you find where the code is “going bad”, start over and this time step INTO the line(s) of code where the problem seems to be originating.
- You should get to a point where you find a single line causing difficulty.
  - This does not mean the current line has a bug - it might highlight a bug elsewhere.
  - It should give you an idea of the problem, so you can find and fix it

What if you don't have a debugger?

Print statements!

- While debugging, you can print out values at any point in the code
  - Print values throughout the code
  - Print explanations of exactly <u>where</u> the print values are from as well.
- By examining the printed values at various points in time, you can see how things are changing in memory, and try to identify buggy code.

Remember

- Treat debugging as a structured process, not random guessing
- Always ensure the bug is repeatable, first
- Isolate it as much as possible
- Then find and fix it
- Test to make sure
  - That the bug is fixed
  - That nothing else was broken
- Think about whether the same error might be elsewhere
  - If so, fix it.

Debugging Outside of Programming

The principles of debugging can apply to dealing with other problems

- E.g. if something is broken on your car.
- Or, the thing you designed/built is not working.

Identify → Isolate → Fix → Test → Other places to fix?

Loops & Repetition

Repetition

One reason we build machines and technology is to make our lives easier.

- And, one way that happens, is to automate repetitive tasks.

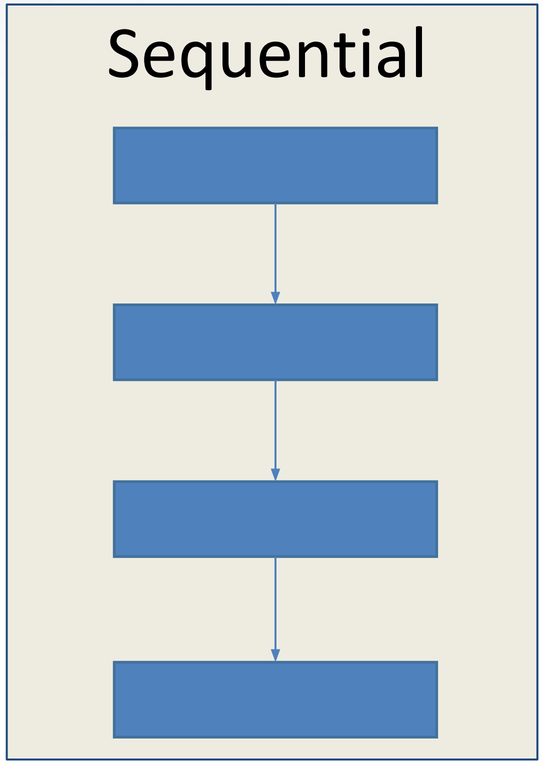
In computer programs, we also have repetitive tasks that we face.

- There are computations that we want to perform over and over
- You’ve seen some, already – times when you wrote a lot of code to do the same thing multiple times.
- You probably had times you wished your program could just “repeat” instead of making it run again

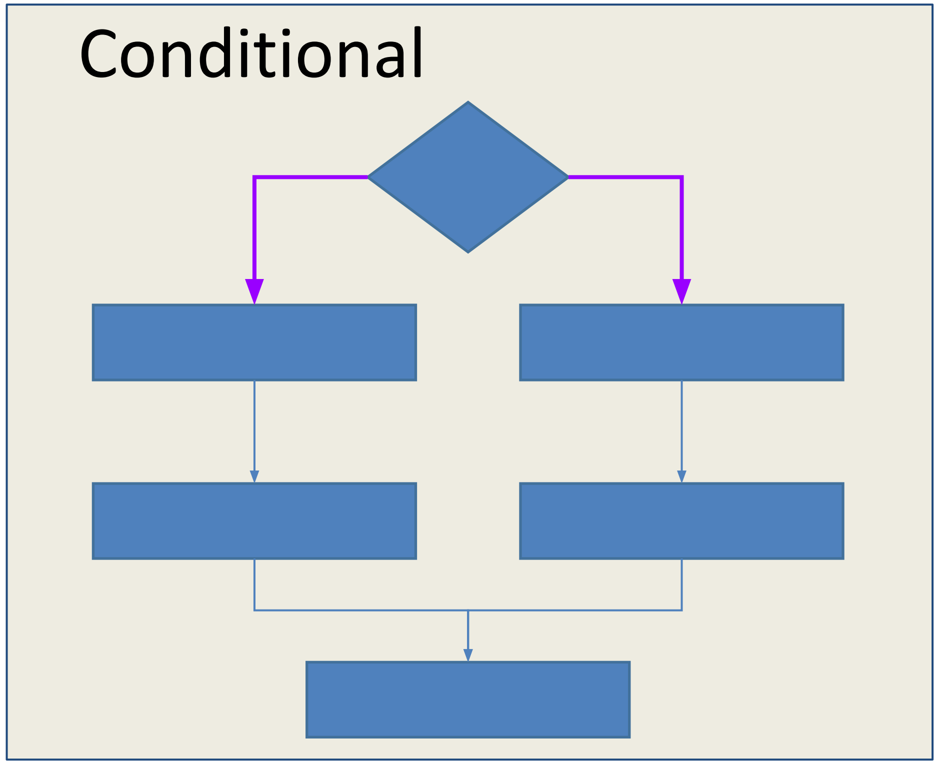
Computational Constructs

A third way of organizing computation (return to earlier point):

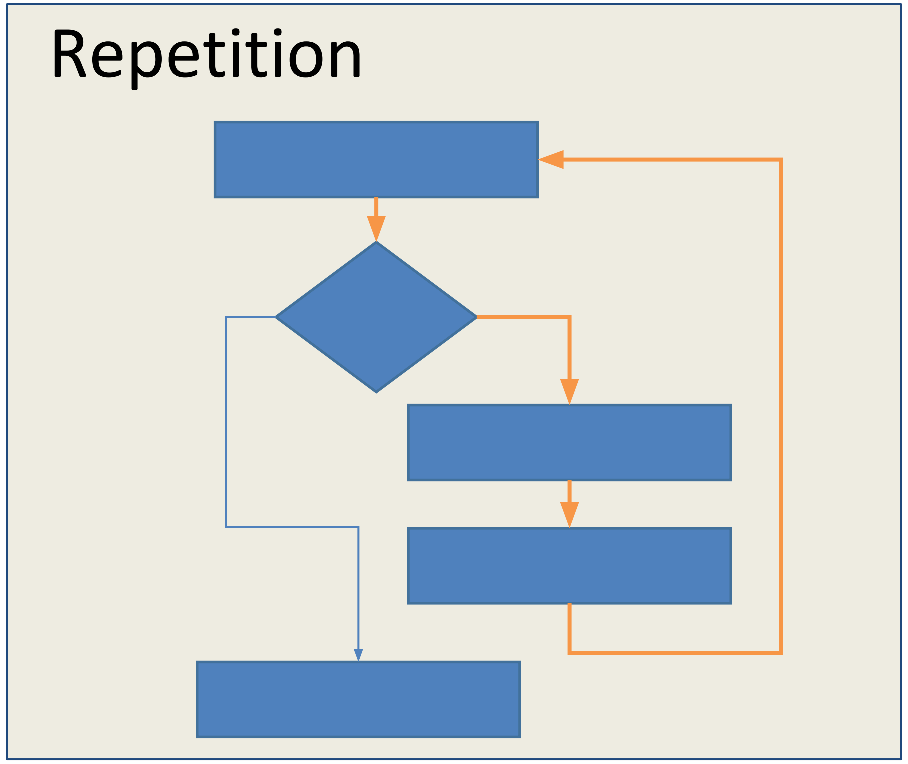
Sequential



Conditional



Repetition



Loops

- Repetition in programs uses a structure called a “loop.”
- There are a few methods, but the key is that the loop will repeat a computation several times.
- We’ll start with the most fundamental loop, the **while** loop

The while statement

"While" statements repeat a set of instructions until a condition is false.

- Format is similar to an *if* statement

Format of a "while" statement:

```
while <condition>:
      <things to do>
```

Start with the keyword while

The while statement

"While" statements repeat a set of instructions until a condition is false.

- Format is similar to an *if* statement

Format of a "while" statement:

```
while <condition>:
    <things to do>
```

The condition is a Boolean that will be evaluated at the beginning of each possible repetition

The while statement

"While" statements repeat a set of instructions until a condition is false.

- Format is similar to an *if* statement

Format of a "while" statement:

```
while <condition>:
    <things to do>
```

Colon at the end

The while statement

“While” statements repeat a set of instructions until a condition is false.

- Format is similar to an *if* statement

Format of a “while” statement:

```
while <condition>:
    <things to do>
```

Then, the commands to repeat are indented (typically 4 spaces)

The while statement

"While" statements repeat a set of instructions until a condition is false.

- Format is similar to an *if* statement

Format of a "while" statement:

```
while <condition>:
    <things to do>
```

The code that is to be repeatedly executed is then included.

How it works

The condition in the while statement is evaluated at the beginning.

- If it is true, then *all* of the indented code is run
- If it is false, then *all* the indented code is skipped

After (all) the indented code is run, then the condition is checked again.

- If the condition is still true, the indented code is run again.
- This can repeat an unlimited number of times!

Note: one execution of the indented code is sometimes called an iteration of the loop.

Example: Guessing Game

ENGINEERING
TEXAS A&M UNIVERSITY

```
print("See if you can guess the number I'm thinking of!")
secret_number = 7
user_guess = int(input("Guess a number from 1 to 10: "))
while user_guess != secret_number:
    print("No! Try again.")
    user_guess = int(input("Guess a number from 1 to 10: "))
print("You guessed it!")
```

Example: Guessing Game

Next

```
print("See if you can guess the number I'm thinking of!")
secret_number = 7
user_guess = int(input("Guess a number from 1 to 10: "))
while user_guess != secret_number:
    print("No! Try again.")
    user_guess = int(input("Guess a number from 1 to 10: "))
print("You guessed it!")
```

Memory

Console

Example: Guessing Game

Next

```
print("See if you can guess the number I'm thinking of!")
secret_number = 7
user_guess = int(input("Guess a number from 1 to 10: "))
while user_guess != secret_number:
    print("No! Try again.")
    user_guess = int(input("Guess a number from 1 to 10: "))
print("You guessed it!")
```

Memory

Console

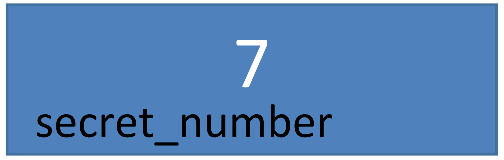
```
See if you can guess the number I'm thinking of!
```

Example: Guessing Game

Next

```
print("See if you can guess the number I'm thinking of!")
secret_number = 7
user_guess = int(input("Guess a number from 1 to 10: "))
while user_guess != secret_number:
    print("No! Try again.")
    user_guess = int(input("Guess a number from 1 to 10: "))
print("You guessed it!")
```

Memory



Console

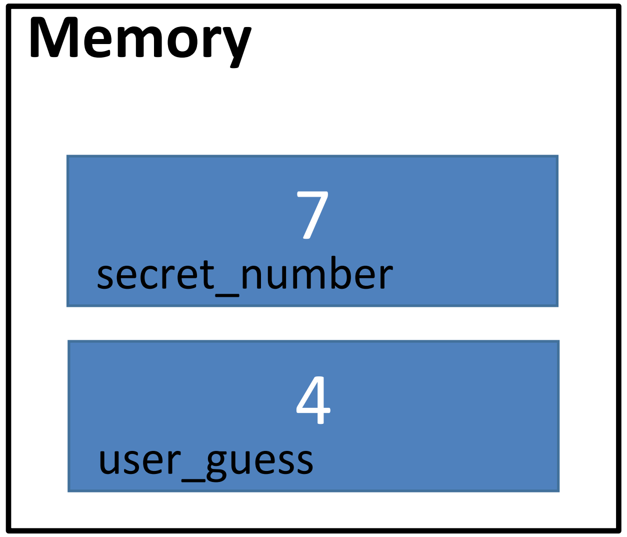
```
See if you can guess the number I'm thinking of!
```

Example: Guessing Game

ENGINEERING
TEXAS A&M UNIVERSITY

Next

```
print("See if you can guess the number I'm thinking of!")
secret_number = 7
user_guess = int(input("Guess a number from 1 to 10: "))
while user_guess != secret_number:
    print("No! Try again.")
    user_guess = int(input("Guess a number from 1 to 10: "))
print("You guessed it!")
```



Console

```
See if you can guess the number I'm thinking of!
Guess a number from 1 to 10: 4
```

The user input the guess “4”

Example: Guessing Game

ENGINEERING
TEXAS A&M UNIVERSITY

```
print("See if you can guess the number I'm thinking of!")
secret_number = 7
user_guess = int(input("Guess a number from 1 to 10: "))
while user_guess != secret_number:
    print("No! Try again.")
    user_guess = int(input("Guess a number from 1 to 10: "))
print("You guessed it!")
```

Next

Memory

7
secret_number

4
user_guess

Console

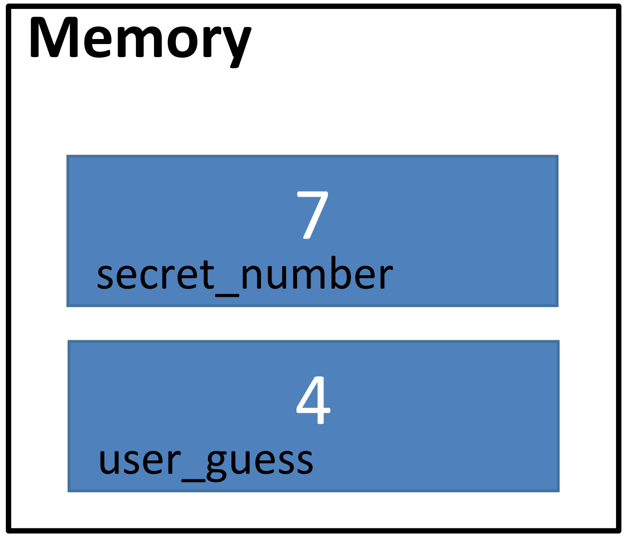
```
See if you can guess the number I'm thinking of!
Guess a number from 1 to 10: 4
```

The condition is True, since user_guess is not secret_number

Example: Guessing Game

```python
print("See if you can guess the number I'm thinking of!")
secret_number = 7
user_guess = int(input("Guess a number from 1 to 10: "))
while user_guess != secret_number:
    print("No! Try again.")
    user_guess = int(input("Guess a number from 1 to 10: "))
print("You guessed it!")
```

Next

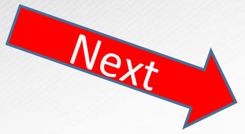


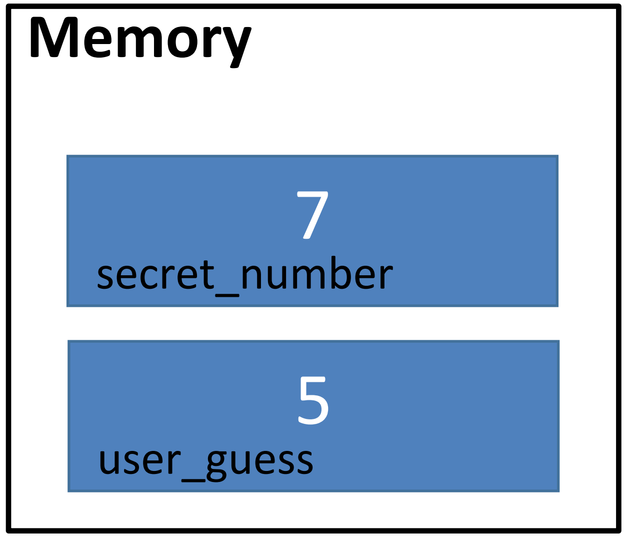
Console

```
See if you can guess the number I'm thinking of!
Guess a number from 1 to 10: 4
No! Try again.
```

Example: Guessing Game

```
print("See if you can guess the number I'm thinking of!")
secret_number = 7
user_guess = int(input("Guess a number from 1 to 10: "))
while user_guess != secret_number:
    print("No! Try again.")
    user_guess = int(input("Guess a number from 1 to 10: "))
print("You guessed it!")
```





Console

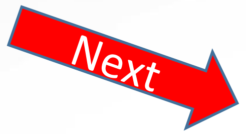
```
See if you can guess the number I'm thinking of!
Guess a number from 1 to 10: 4
No! Try again.
Guess a number from 1 to 10: 5
```

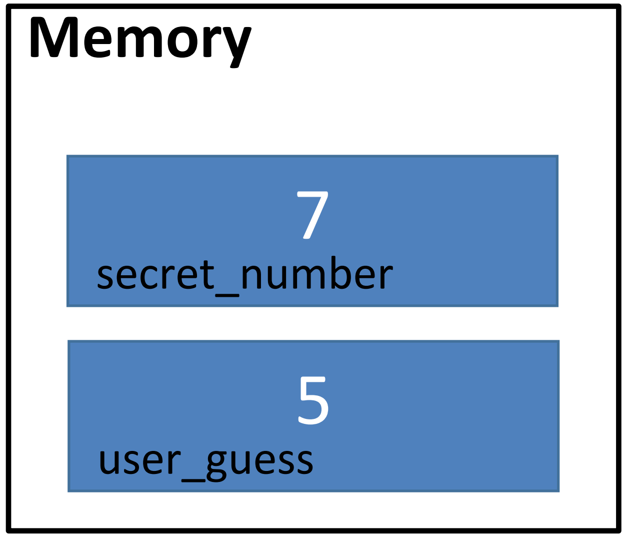
The user input the guess “5”.

We’re at the end of the indented block, so we check the condition again.

Example: Guessing Game

```
print("See if you can guess the number I'm thinking of!")
secret_number = 7
user_guess = int(input("Guess a number from 1 to 10: "))
while user_guess != secret_number:
    print("No! Try again.")
    user_guess = int(input("Guess a number from 1 to 10: "))
print("You guessed it!")
```





Console

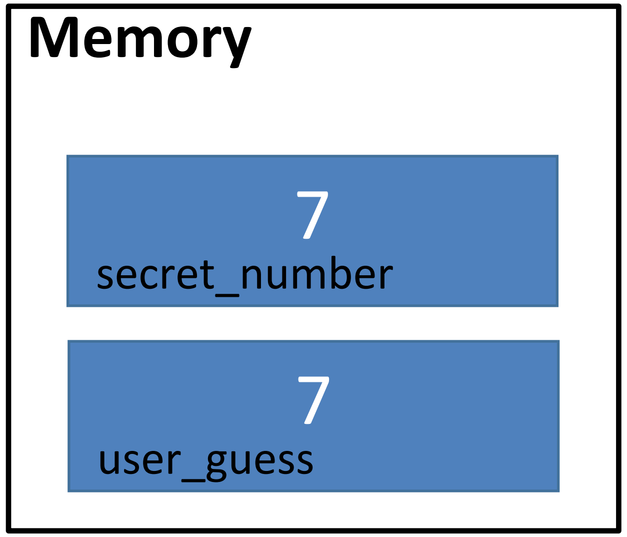
```
See if you can guess the number I'm thinking of!
Guess a number from 1 to 10: 4
No! Try again.
Guess a number from 1 to 10: 5
No! Try again.
```

Example: Guessing Game

ENGINEERING
TEXAS A&M UNIVERSITY

Next

```
print("See if you can guess the number I'm thinking of!")
secret_number = 7
user_guess = int(input("Guess a number from 1 to 10: "))
while user_guess != secret_number:
    print("No! Try again.")
    user_guess = int(input("Guess a number from 1 to 10: "))
print("You guessed it!")
```



Console

```
See if you can guess the number I'm thinking of!
Guess a number from 1 to 10: 4
No! Try again.
Guess a number from 1 to 10: 5
No! Try again.
Guess a number from 1 to 10: 7
```

The user input the guess “7”.

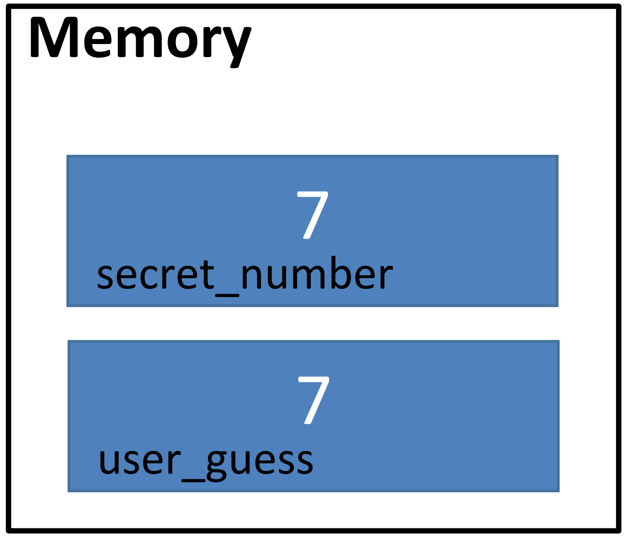
We’re at the end of the indented block, so we check the condition again.

Example: Guessing Game

ENGINEERING
TEXAS A&M UNIVERSITY

```
print("See if you can guess the number I'm thinking of!")
secret_number = 7
user_guess = int(input("Guess a number from 1 to 10: "))
while user_guess != secret_number:
    print("No! Try again.")
    user_guess = int(input("Guess a number from 1 to 10: "))
print("You guessed it!")
```

Next



Console

```
See if you can guess the number I'm thinking of!
Guess a number from 1 to 10: 4
No! Try again.
Guess a number from 1 to 10: 5
No! Try again.
Guess a number from 1 to 10: 7
You guessed it!
```

All done!

Exercise

See if you can modify the program to count and print the number of guesses it took the user to guess the secret number

```
print("See if you can guess the number I'm thinking of!")
secret_number = 7
user_guess = int(input("Guess a number from 1 to 10: "))
while user_guess != secret_number:
    print("No! Try again.")
    user_guess = int(input("Guess a number from 1 to 10: "))
print("You guessed it!")
```

Exercise

See if you can modify the program to count and print the number of guesses it took the user to guess the secret number

```
print("See if you can guess the number I'm thinking of!")
secret_number = 7
user_guess = int(input("Guess a number from 1 to 10: "))
num_guesses = 1
while user_guess != secret_number:
    print("No! Try again.")
    user_guess = int(input("Guess a number from 1 to 10: "))
    num_guesses += 1
print("You guessed it! It took you", num_guesses, "guesses.")
```

Infinite loops

It's possible for a loop to go on forever!

- We call this an infinite loop

```
while True:
    print("Here we go again.")
```

If your code is stuck in an infinite loop, you'll want to "break" out of it.

- in PyCharm: use stop execution (the red square in PyCharm)
- in a separate window: usually ctrl-c will stop the program.

Checking conditions

What will happen with the following code?

```
a = 0
while a == 0:
    print("About to change a")
    a = 1
    print("Changing a back to 0")
    a = 0
```

Checking conditions

What will happen with the following code?

```
a = 0
while a == 0:
    print("About to change a")
    a = 1
    print("Changing a back to 0")
    a = 0
```

Console

```
About to change a
Changing a back to 0
About to change a
Changing a back to 0
...
```

An infinite loop!!

Checking conditions

Changing “a” in the middle of the loop does not matter

- The condition is only evaluated after all the indented code is run
- So, in this case, a is 0 every time the condition is checked

A common loop

One of the most common loops used in programming is where a loop runs a certain number of times - basically like this:

```
i = 0
while i < 10:
    print("Doing something")
    i += 1
```

A common loop

One of the most common loops used in programming is where a loop runs a <u>certain</u> number of times - basically like this:

```
i = 0
while i < 10:
    print("Doing something")
    i += 1
```

We keep a variable that counts which iteration we are on, usually from 0 (not 1).

The variable name "i" (as well as sometimes "j" and "k") is commonly used for counting in this way.

ENGINEERING
TEXAS A&M UNIVERSITY

A common loop

One of the most common loops used in programming is where a loop runs a certain number of times - basically like this:

```
i = 0
while i < 10:
    print("Doing something")
    i += 1
```

The condition compares the counter (i) to the number of loop iterations we want (10 in this case).

We could have used a variable to store the number of iterations.

A common loop

One of the most common loops used in programming is where a loop runs a <u>certain</u> number of times - basically like this:

```
i = 0
while i < 10:
    print("Doing something")
    i += 1
```

Inside the loop, we will do something. This could be lots of lines of code, performing calculations, etc.

A common loop

One of the most common loops used in programming is where a loop runs a certain number of times - basically like this:

```
i = 0
while i < 10:
    print("Doing something")
    i += 1
```

Finally, we increment the counter (i)

A common loop

One of the most common loops used in programming is where a loop runs a certain number of times - basically like this:

```
i = 0
while i < 10:
    print("Doing something")
    i += 1
```

We will execute this loop a total of 10 times - thus printing out “Doing something” 10 times.

A common loop

What if we want to modify the loop so that i goes from 1 to 10 instead of 0 to 9?

```
i = 1
while i <= 10:
    print("Doing something")
    i += 1
```

Note: it is better to write i <= 10 rather than i < 11, since the number of iterations (10) is more clearly stated.

The for loop

The loop just described is so common that another loop structure is defined to handle cases like that!

The “**for**” loop.

- For loops can have a few variations, but the main one we’ll look at corresponds to the loop just shown

For loop structure

```
for i in range(10):
    print("Doing Something")
```

Begin with the keyword “for”

For loop structure

```
for i in range(10):
    print("Doing Something")
```

We give the variable name that will take on all the different values.

We call i an “iterator”

ENGINEERING
TEXAS A&M UNIVERSITY

For loop structure

```
for i in range(10):
    print("Doing Something")
```

Next we have the keyword “in”

For loop structure

```
for i in range(10):
    print("Doing Something")
```

And, next, we have the range command.

This says that the values to be taken on are those in the range from 0 to 9

The range is 0 to 9, not 1 to 10!

For loop structure

```
for i in range(10):
    print("Doing Something")
```

There is a colon at the end of the line.

ENGINEERING
TEXAS A&M UNIVERSITY

For loop structure

```
for i in range(10):
    print("Doing Something")
```

Then an indentation.

For loop structure

```
for i in range(10):
    print("Doing Something")
```

And then the command(s) to be done on each iteration.
In this case, it’s just a single print statement.

Equivalent loops

So, these two loops are equivalent to each other

```
i = 0
while i < 10:
    print("Doing something")
    i += 1
```

```
for i in range(10):
    print("Doing Something")
```

You’ll get this later…

The range(10) command is creating a **list** containing the numbers 0 to 9.

- Then, the for loop is letting i take on each value in that list.

Exercise

Say you wanted to sum up the numbers from 1 to 10. How would you do that with a loop?

Exercise

Say you wanted to sum up the numbers from 1 to 10. How would you do that with a loop?

```
sum = 0
for i in range(10):
    sum += i+1
print(sum)
```

- You have to add "i+1" each time, since the values of i are 0 to 9, not 1 to 10.
- *There's also a formula for summing up numbers from 1 to n: n(n+1)/2*

ENGINEERING
TEXAS A&M UNIVERSITY

Nesting Loops

Like conditionals, you can **nest** loops

Example: print out a list of numbers from 0-9 multiplied by all numbers from 0-9:

```
for i in range(10):
    for j in range(10):
        print (i, "times", j, "equals", i*j)
```

Loops in programming

When to use a **for** loop:

- When you have a known number of iterations
- When you want to iterate through a specific, known set of items

When to use a **while** loop:

- When you want to repeat indefinitely
- When you are repeating until a specific value is encountered
- When you want to repeat until a general condition is met

Current Assignments

Complete the ECEN module and Academic Honesty part 2 modules on the eCommunity page

- Due 9/29, by end-of-day

Lab Assignment 5	Due 9/29 by end-of-day
Lab Assignment 5b	Due 9/29 by end-of-day

Preactivity: Complete zyBook chapter 7.1–7.7 *(excluding 7.2)* prior to class next week