ISE 314X
Computer Programing for Engineers

Chapter 1
Computers and Programs

Yong Wang
Assistant Professor
Systems Science & Industrial Engineering
Binghamton University

Objectives

- To understand the roles of hardware and software in a computing system
- To understand the functions of computer programming languages
- To begin using Python

Computer Programs

- *A detailed, step-by-step set of instructions telling a computer what to do*
- If we change the program, the computer performs a different set of actions

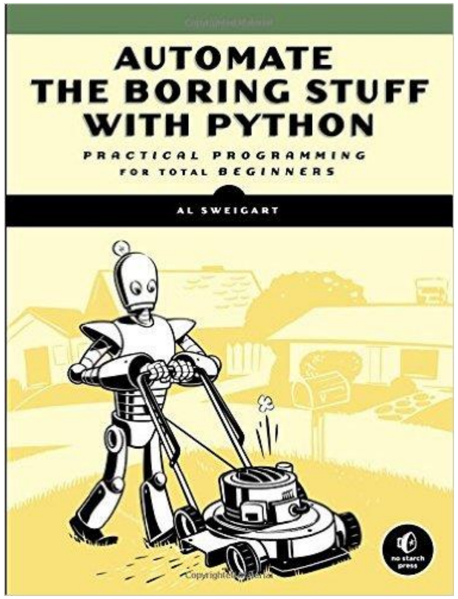
Computer Programs

- *Software* (programs) rule the *hardware* (the physical machine)
- The process of creating software is called *programming*

Why Learn to Program?

- Helps you become a more intelligent user of computers
- Automate the boring stuff
- Improve problem solving skills
- Programmers are in great

  demand in various industries



Problem Solving

- Some problems can be *solved*. This is done by developing an *algorithm*, a step-by-step description for achieving the desired result
- **What is the difference between an algorithm and a program?**

Problem Solving

- Some problems are not solvable by any algorithm. These problems are said to be *unsolvable* (e.g., the Halting Problem)
- Due to hardware limitations, some problems can be *intractable* if they would take too long or take too much memory to be of practical value (e.g., Large OR problems)

CPU

- The *central processing unit* (CPU) is the "brain" of a computer
- It carries out all the operations on the data
- Examples
  - – arithmetic operations: +, -, *, /
  - – logical operations: testing if two numbers are equal

Memory

- Memory stores programs and data
  - CPU can only directly access information stored in *main memory* (RAM or Random Access Memory)
  - Main memory is fast, but *volatile*, i.e. when the power is interrupted, the contents of memory are lost
  - *Secondary memory* provides more permanent storage: magnetic (hard drive, floppy), optical (CD, DVD)

I/O Devices

- *Input* devices
  - Information is passed to the computer through keyboards, mice, touchscreens, microphones, etc.
- *Output* devices
  - Processed information is presented to the user through the display, printer, speaker, etc.

CPU Execution Cycle

- *Fetch, Execute, Repeat*
  - An instruction retrieved from memory
  - Decode the instruction to see what it represents
  - Appropriate action carried out
  - Repeat until no more instructions in the memory

Programming Languages

- Natural languages are ambiguous and imprecise
- Programs are expressed in an unambiguous, precise way using *programming languages*

Programming Languages

- Every structure in programming language has a precise form, called its *syntax*
- Every structure in programming language has a precise meaning, called its *semantics*

Programming & Coding

- Programmers often refer to their program as *computer code*
- Process of writing an algorithm in a programming language often called *coding*

High-Level vs. Low-Level Programming Languages

- *High-level* computer languages
  - Designed to be used and understood by humans
  - Examples: fortran, c, c++, pascal, java, basic, python, matlab, R
- *Low-level* language
  - Computer hardware can only understand a very low level language known as *machine language*
  - Examples: assembly language, machine code (0's and 1's)

Example: Adding Two Numbers

- Low-level language
  - Load the number **a** from memory location 5107 into the CPU
  - Load the number **b** from memory location 5108 into the CPU
  - Add the two numbers in the CPU and assign it to **c**
  - Store **c** into location 5109

Example: Adding Two Numbers

- High-level language

  c = a + b

Compiled Languages

- High-level languages can be divided into two types: *compiled and interpreted*
- *Compilers* convert programs written in a high-level language into the machine language
- Once program is compiled, it can be executed over and over without the source code or compiler
- Examples: fortran, c, c++, pascal, java

Interpreted Languages

- The source program is not translated into machine language all at once
- An *interpreter* analyses and executes the source code line by line
- Examples: python, matlab, R, Visual basic

Compiled vs Interpreted

- Compiled programs run faster since the translation of the source code happens only once
- Interpreted programs are more flexible and take less time to develop

Questions

- **Compiled language vs low-level languages, which is faster?**
- **Why do we teach Python (an interpreted language) over a compiled language?**

Get Started with Python

- When you start Python in IDLE, you will see something like this:

```
Python 3.5.2 |Anaconda 4.1.1 (64-bit)|
  (default, Aug  1 2016, 11:39:45) [MSC v.1600
  64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or
  "license" for more information.
>>>
```

Get Started with Python

- The “>>>” is a Python *prompt* indicating that Python is ready for us to give it a command
- In python, each command is called a *statement*

```
>>> print("Hello, world!")
Hello, world!
>>> 2 + 3
5
>>> print("2 + 3 =", 2 + 3)
2 + 3 = 5
```

Get Started with Python

- Usually we want to execute several statements together to solve a problem
- One way to do this is to use a *function*

```
>>> def hello():
...     print("Hello,")
...     print("world!")

...
>>>
```

Get Started with Python

```
>>> def hello():
...     print("Hello,")
...     print("world!")
...
>>>
```

- The first line tells Python we are *defining* a new function called hello
- The following lines are *indented with four spaces* to show that they are part of the hello function
- The blank line (hit enter twice) lets Python know the definition is finished

Get Started with Python

```
>>> def hello():
...     print("Hello,")
...     print("world!")
...
>>>
```

- A function is *invoked* by typing its name

```
>>> hello()
Hello,
World!
```

Get Started with Python

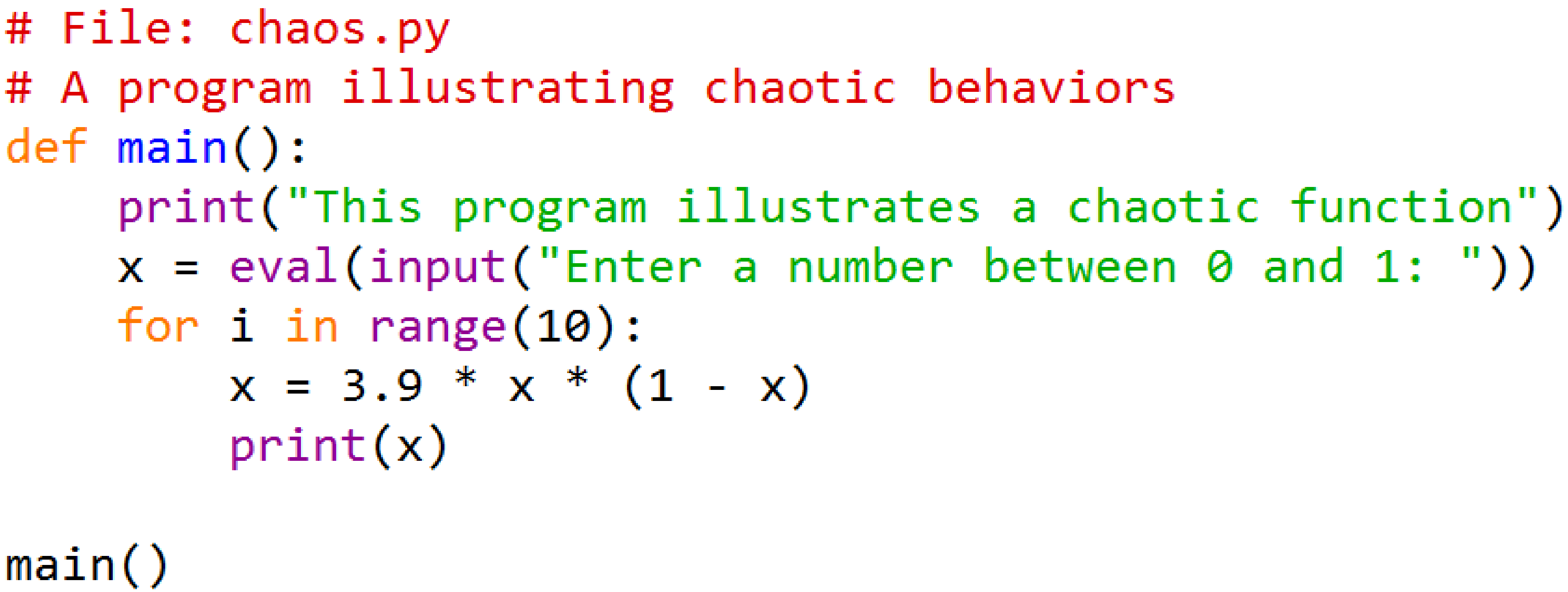
- Commands can have changeable parts called *parameters* that are placed between the ()’s.

```
>>> def greet(User):
...     print("Hello,", User)
...
>>> greet("Terry")
Hello, Terry
>>> greet("Paul")
Hello, Paul
```

Python Modules

- When we exit the Python prompt, the functions we've defined cease to exist!
- Programs are usually composed of functions, *modules*, or *scripts* that are saved on disk so that they can be used again and again
- A *module file* is a text file created in a plain-text editor that contains function definitions

chaos.py



```
# File: chaos.py
# A program illustrating chaotic behaviors
def main():
    print("This program illustrates a chaotic function")
    x = eval(input("Enter a number between 0 and 1: "))
    for i in range(10):
        x = 3.9 * x * (1 - x)
        print(x)

main()
```

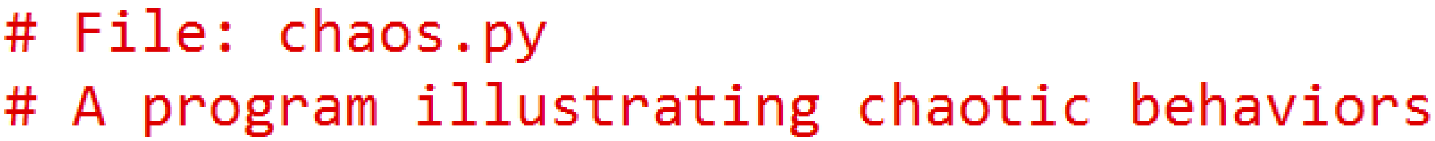
- We'll use *.py* when we save our work to indicate it's a Python program
- In this code we're defining a new function called *main*

Run chaos.py

- Open the file in IDLE and Run:

```
This program illustrates a chaotic function
Enter a number between 0 and 1: 0.5
0.975
0.09506250000000008
0.3354999226562525
0.8694649252590003
0.44263310911310905
0.962165255336889
0.1419727793616139
0.4750843861996143
0.9725789275369049
0.1040097132674683
```

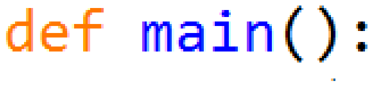
Inside a Python Program



```
# File: chaos.py
# A program illustrating chaotic behaviors
```

- Lines that start with # are called *comments*
- Intended for human readers
- Python skips text from # to end of line

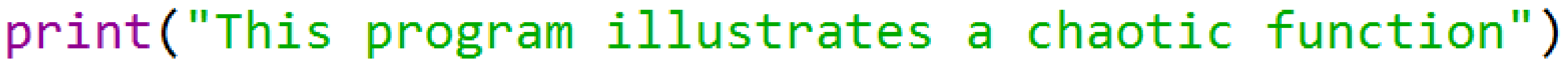
Inside a Python Program



```
def main():
```

- Beginning of the definition of a function called *main*
- This program has only one module. It could have been written without the *main* function
- The use of *main* is customary, however

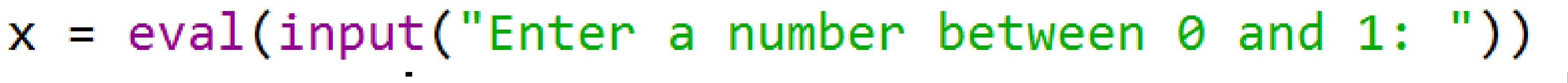
Inside a Python Program



```
print("This program illustrates a chaotic function")
```

- This line causes Python to print a message introducing the program

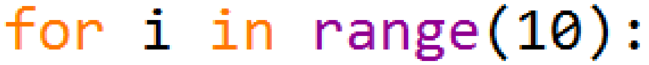
Inside a Python Program



```
x = eval(input("Enter a number between 0 and 1: "))
```

- A variable is used to assign a name to a value so that we can refer to it later
- The quoted information is displayed, and the number typed in response is stored in x

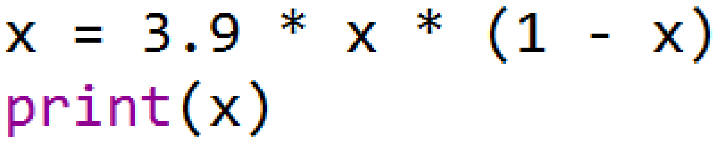
Inside a Python Program



```
for i in range(10):
```

- for is a *loop* keyword
- A loop tells Python to repeat the same thing over and over
- In this example, the body of the loop will be repeated 10 times

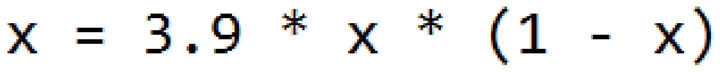
Inside a Python Program



```
x = 3.9 * x * (1 - x)
print(x)
```

- These lines are the *body* of the loop
- The body of the loop is identified through *indentation*
- The effect of the loop is the same as repeating this two lines 10 times!

Inside a Python Program



```
x = 3.9 * x * (1 - x)
```

- This is called an *assignment* statement
- The part on the right-hand side (RHS) of the "=" is a mathematical expression
- * is used to indicate multiplication
- Once the value on the RHS is computed, it is stored back into x

Inside a Python Program

```
main()
```

- This last line tells Python to *execute* the code in the function *main*

Chaos

- The function in this program has the general form $x_{n+1} = k \cdot x_n \cdot (1 - x_n)$, where *k* is 3.9
- This type of function is known as a *logistic function*
- *Chaotic behaviors*: Very small differences in initial value can make large differences in the output

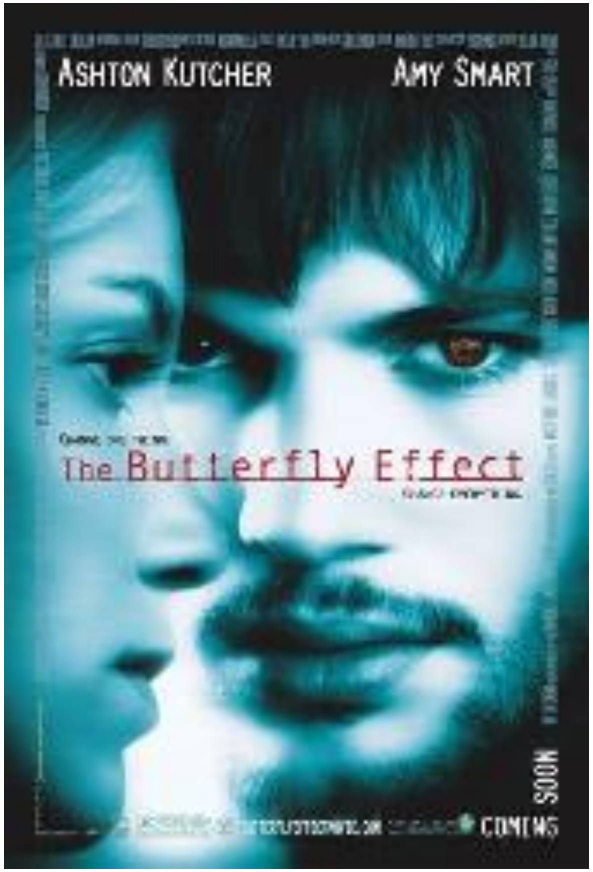
Chaos

```
Enter a number
between 0 and 1: 0.25
0.73125
0.76644140625
0.698135010439
0.82189581879
0.570894019197
0.955398748364
0.166186721954
0.540417912062
0.9686289303
0.118509010176
```

```
Enter a number
between 0 and 1: 0.26
0.75036
0.73054749456
0.767706625733
0.6954993339
0.825942040734
0.560670965721
0.960644232282
0.147446875935
0.490254549376
0.974629602149
```

Chaos & Butterfly Effect

- Computer models that are used to simulate and predict weather patterns are very sensitive
- A butterfly flapping its wings in LA might affect whether it will rain in NYC



Chaos & Butterfly Effect

- Factors like this are just too many to be included in the computer model
- We can only make predictions for a few days in advance
- Accurate prediction over a longer time frame is unachievable