CS 115 Lecture 2

Fundamentals of computer science, computers, and programming

Neil Moore

Department of Computer Science
University of Kentucky
Lexington, Kentucky 40506
`neil@cs.uky.edu`

1 September 2015

What is programming?

CS 115 is titled “Introduction to Computer Programming”. What is that?

What is programming?

CS 115 is titled “Introduction to Computer Programming”. What is that?

- Telling a computer what to do?
  - But every time I click on a button, or press a key,
    I am telling the computer what to do.
  - That’s not quite what we mean by programming.

What is programming?

CS 115 is titled "Introduction to Computer Programming". What is that?

- Telling a computer what to do?
  - But every time I click on a button, or press a key,
    I am telling the computer what to do.
  - That's not quite what we mean by programming.
- Writing computer programs.

What is programming?

CS 115 is titled “Introduction to Computer Programming”. What is that?

- Telling a computer what to do?
  - But every time I click on a button, or press a key, I am telling the computer what to do.
  - That’s not quite what we mean by programming.
- Writing computer programs.
  - What’s that?
  - What is a “program” outside of computing?

What is programming?

CS 115 is titled "Introduction to Computer Programming". What is that?

- Telling a computer what to do?
    - But every time I click on a button, or press a key, I am telling the computer what to do.
    - That's not quite what we mean by programming.
- Writing computer programs.
    - What's that?
    - What is a "program" outside of computing?
        - TV show.

What is programming?

CS 115 is titled “Introduction to Computer Programming”. What is that?

- Telling a computer what to do?
    - But every time I click on a button, or press a key, I am telling the computer what to do.
    - That’s not quite what we mean by programming.
- Writing computer programs.
    - What’s that?
    - What is a “program” outside of computing?
        - TV show.
        - Concert program

What is programming?

CS 115 is titled “Introduction to Computer Programming”. What is that?

- Telling a computer what to do?
  - But every time I click on a button, or press a key,
    I am telling the computer what to do.
  - That’s not quite what we mean by programming.
- Writing computer programs.
  - What’s that?
  - What is a “program” outside of computing?
    - ⋆ TV show.
    - ⋆ Concert program: what is going to happen, in what order.

What is programming?

CS 115 is titled “Introduction to Computer Programming”. What is that?

- Telling a computer what to do?
  - But every time I click on a button, or press a key,
    I am telling the computer what to do.
  - That’s not quite what we mean by programming.
- Writing computer programs.
  - What’s that?
  - What is a “program” outside of computing?
    - TV show.
    - Concert program: what is going to happen, in what order.
  - A sequence of instructions telling a computer how to do something.

What is programming?

CS 115 is titled "Introduction to Computer Programming". What is that?

- Telling a computer what to do?
  - But every time I click on a button, or press a key,
    I am telling the computer what to do.
  - That's not quite what we mean by programming.
- Writing computer programs.
  - What's that?
  - What is a "program" outside of computing?
    - TV show.
    - Concert program: what is going to happen, in what order.
  - A sequence of instructions telling a computer how to do something.
  - Plan out in advance how to solve a kind of problem.
    - Then we have the computer execute the program

What is programming?

CS 115 is titled “Introduction to Computer Programming”. What is that?

- Telling a computer what to do?
  - But every time I click on a button, or press a key,
    I am telling the computer what to do.
  - That’s not quite what we mean by programming.
- Writing computer programs.
  - What’s that?
  - What is a “program” outside of computing?
    - TV show.
    - Concert program: what is going to happen, in what order.
  - A sequence of instructions telling a computer how to do something.
  - Plan out in advance how to solve a kind of problem.
    - Then we have the computer execute the program

What is computer science?

So what is computer science, and how does it differ from programming?

What is computer science?

So what is computer science, and how does it differ from programming?

- “The study of computers”?

What is computer science?

So what is computer science, and how does it differ from programming?

- "The study of computers"?
  - "Computer science is no more about computers than astronomy is about telescopes." –attributed to Edsgar Dijkstra.

What is computer science?

So what is computer science, and how does it differ from programming?

- "The study of computers"?
  - "Computer science is no more about computers than astronomy is about telescopes." –attributed to Edsgar Dijkstra.
- Questions about computation came up long before computers.
  - It used to be *people* following the step-by-step instructions.
    - ⋆ Abacus, slide rule, pencil and paper, . . .
  - What did we call those people?

What is computer science?

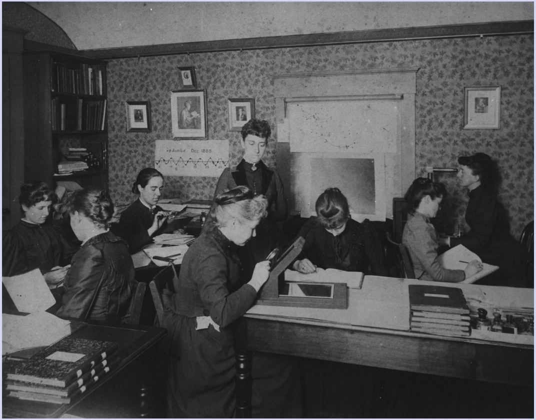
So what is computer science, and how does it differ from programming?

- "The study of computers"?
    - "Computer science is no more about computers than astronomy is about telescopes." –attributed to Edsgar Dijkstra.
- Questions about computation came up long before computers.
    - It used to be *people* following the step-by-step instructions.
        - ⋆ Abacus, slide rule, pencil and paper, . . .
    - What did we call those people? "Computers"

What is computer science?

So wha ng?

- “ is
- Q



An early computer network, around 1890.

E.C. Pickering’s astronomy lab at Harvard.

Image: Harvard University, Wikipedia article “Harvard Computers”

What is computer science?

So what is computer science, and how does it differ from programming?

- "The study of computers"?
  - "Computer science is no more about computers than astronomy is about telescopes." –attributed to Edsgar Dijkstra.
- Questions about computation came up long before computers.
  - It used to be *people* following the step-by-step instructions.
    - ⋆ Abacus, slide rule, pencil and paper, . . .
  - What did we call those people? "Computers"
  - When you do long division or sort a list of names, you are computing.

What is computer science?

So what is computer science, and how does it differ from programming?

- "The study of computers"?
  - "Computer science is no more about computers than astronomy is about telescopes." –attributed to Edsgar Dijkstra.
- Questions about computation came up long before computers.
  - It used to be *people* following the step-by-step instructions.
    - Abacus, slide rule, pencil and paper, ...
  - What did we call those people? "Computers"
  - When you do long division or sort a list of names, you are computing.
- Computer science is the study of:
  - What can be computed using step-by-step procedures.
  - How best to specify these procedures.
  - How to tell if a procedure is correct, efficient, etc.
  - How to design procedures to solve real-world problems.

Algorithms

“Step-by-step procedure” is a mouthful.
We have a name for that: an **algorithm**.

Algorithms

“Step-by-step procedure” is a mouthful.
We have a name for that: an **algorithm**.

- A “well-ordered collection of unambiguous and effectively computable operations that when executed produces a result and halts in a finite amount of time.” [Schneider and Gersting].

Algorithms

"Step-by-step procedure" is a mouthful.
We have a name for that: an **algorithm**.

- A "well-ordered collection of unambiguous and effectively computable operations that when executed produces a result and halts in a finite amount of time." [Schneider and Gersting].
- Named after the 9th-century Persian mathematician Muhammad ibn Musa al-Khwarizmi.

Algorithms

"Step-

We ha

- A ... ıtable
  o ... inite
  ar
- N
  M

Muhammad Al-Khwarizmi, Persian computer scientist.

Sculpture: S. Ch. Babajan / Image: Michael Zaretski, Flickr.

Algorithms

"Step-by-step procedure" is a mouthful.
We have a name for that: an **algorithm**.

- A "well-ordered collection of unambiguous and effectively computable operations that when executed produces a result and halts in a finite amount of time." [Schneider and Gersting].
- Named after the 9th-century Persian mathematician Muhammad ibn Musa al-Khwarizmi.
  - "The Compendious Book on Calculation by Restoring and Balancing"
  - Described how to solve linear and quadratic equations.

Algorithms

"Step-by-step procedure" is a mouthful.
We have a name for that: an **algorithm**.

- A "well-ordered collection of unambiguous and effectively computable operations that when executed produces a result and halts in a finite amount of time." [Schneider and Gersting].
- Named after the 9th-century Persian mathematician Muhammad ibn Musa al-Khwarizmi.
  - "The Compendious Book on Calculation by Restoring and Balancing"
  - Described how to solve linear and quadratic equations.
  - Arabic: *Al-kitab al-mukhtasar fi hisab* **al-jabr** *wa'l-muqabala*

Algorithms

"Step-by-step procedure" is a mouthful.
We have a name for that: an **algorithm**.

- A "well-ordered collection of unambiguous and effectively computable operations that when executed produces a result and halts in a finite amount of time." [Schneider and Gersting].
- Named after the 9th-century Persian mathematician Muhammad ibn Musa al-Khwarizmi.
  - "The Compendious Book on Calculation by Restoring and Balancing"
  - Described how to solve linear and quadratic equations.
  - Arabic: *Al-kitab al-mukhtasar fi hisab* **al-jabr** *wa'l-muqabala*
    - "Algebra"

Algorithms

"Step-by-step procedure" is a mouthful.
We have a name for that: an **algorithm**.

- A "well-ordered collection of unambiguous and effectively computable operations that when executed produces a result and halts in a finite amount of time." [Schneider and Gersting].
- Named after the 9th-century Persian mathematician Muhammad ibn Musa al-Khwarizmi.
    - "The Compendious Book on Calculation by Restoring and Balancing"
    - Described how to solve linear and quadratic equations.
    - Arabic: *Al-kitab al-mukhtasar fi hisab* **al-jabr** *wa'l-muqabala*
        - "Algebra"
- Let's look at one algorithm that is even older than that:
    - Euclid's greatest common divisor algorithm.
    - One of the oldest algorithms that is still in use.
        - In Euclid's *Elements*, written around 300 BCE.
        - Older than long division!

Algorithms

"Step-by-step procedure" is a mouthful.
We have a name for that: an **algorithm**.

- A "well-ordered collection of unambiguous and effectively computable operations that when executed produces a result and halts in a finite amount of time." [Schneider and Gersting].
- Named after the 9th-century Persian mathematician Muhammad ibn Musa al-Khwarizmi.
  - "The Compendious Book on Calculation by Restoring and Balancing"
  - Described how to solve linear and quadratic equations.
  - Arabic: *Al-kitab al-mukhtasar fi hisab* **al-jabr** *wa'l-muqabala*
    - ⋆ "Algebra"
- Let's look at one algorithm that is even older than that:
  - Euclid's greatest common divisor algorithm.
  - One of the oldest algorithms that is still in use.
    - ⋆ In Euclid's *Elements*, written around 300 BCE.
    - ⋆ Older than long division!

Euclid's algorithm

Given two numbers *a* and *b*, their **greatest common divisor** (GCD) is the largest number that both are divisible by.

- The GCD of 10 and 25 is 5; of 13 and 3 is 1.

Euclid's algorithm

Given two numbers a and b, their **greatest common divisor** (GCD) is the largest number that both are divisible by.

- The GCD of 10 and 25 is 5; of 13 and 3 is 1.

Euclid designed an algorithm to compute the GCD:

Inputs: two positive integers (whole numbers) a and b.

Outputs: the GCD of a and b

Euclid's algorithm

Given two numbers *a* and *b*, their **greatest common divisor** (GCD) is the largest number that both are divisible by.

- The GCD of 10 and 25 is 5; of 13 and 3 is 1.

Euclid designed an algorithm to compute the GCD:

Inputs: two positive integers (whole numbers) *a* and *b*.

Outputs: the GCD of *a* and *b*

```
1 Repeat as long as $b$ is not zero:
    1.1 If $a > b$, then set $a \leftarrow (a - b)$.
    1.2 Otherwise, set $b \leftarrow (b - a)$.
```

Euclid's algorithm

Given two numbers a and b, their **greatest common divisor** (GCD) is the largest number that both are divisible by.

- The GCD of 10 and 25 is 5; of 13 and 3 is 1.

Euclid designed an algorithm to compute the GCD:

Inputs: two positive integers (whole numbers) a and b.

Outputs: the GCD of a and b

```
1 Repeat as long as $b$ is not zero:
    1.1 If $a > b$, then set $a \leftarrow (a - b)$.
    1.2 Otherwise, set $b \leftarrow (b - a)$.
2 Output $a$ as the answer.
```

Euclid's algorithm

Given two numbers a and b, their **greatest common divisor** (GCD) is the largest number that both are divisible by.

- The GCD of 10 and 25 is 5; of 13 and 3 is 1.

Euclid designed an algorithm to compute the GCD:

```
Inputs: two positive integers (whole numbers) $a$ and $b$.
Outputs: the GCD of $a$ and $b$
1 Repeat as long as $b$ is not zero:
    1.1 If $a > b$, then set $a \leftarrow (a - b)$.
    1.2 Otherwise, set $b \leftarrow (b - a)$.
2 Output $a$ as the answer.
```

Euclid proved that this algorithm is **correct** (it gives the right answer) and **effective** (it always gives an answer).

Euclid's algorithm

Given two numbe (GCD) is the
largest number t

- The GCD o

Euclid designed a

Inputs: two

Outputs: th

1. Repeat as lo
   1.1 If $a > b$
   1.2 Otherwi
2. Output a as

Euclid proved th : answer) and
effective (it alw

Euclid, Greek computer scientist.

Sculpture: J. Durham / Image: Mark A. Wilson, Wikipedia, 2005.

Euclid's algorithm

Given two numbers a and b, their **greatest common divisor** (GCD) is the largest number that both are divisible by.

- The GCD of 10 and 25 is 5; of 13 and 3 is 1.

Euclid designed an algorithm to compute the GCD:

```
Inputs: two positive integers (whole numbers) $a$ and $b$.
Outputs: the GCD of $a$ and $b$
1 Repeat as long as $b$ is not zero:
    1.1 If $a > b$, then set $a \leftarrow (a - b)$.
    1.2 Otherwise, set $b \leftarrow (b - a)$.
2 Output $a$ as the answer.
```

Euclid proved that this algorithm is **correct** (it gives the right answer) and **effective** (it always gives an answer). Let's try a few examples.

Euclid's algorithm

Given two numbers *a* and *b*, their **greatest common divisor** (GCD) is the largest number that both are divisible by.

- The GCD of 10 and 25 is 5; of 13 and 3 is 1.

Euclid designed an algorithm to compute the GCD:

```
Inputs: two positive integers (whole numbers) $a$ and $b$.
Outputs: the GCD of $a$ and $b$
1 Repeat as long as $b$ is not zero:
    1.1 If $a > b$, then set $a \leftarrow (a - b)$.
    1.2 Otherwise, set $b \leftarrow (b - a)$.
2 Output $a$ as the answer.
```

Euclid proved that this algorithm is **correct** (it gives the right answer) and **effective** (it always gives an answer). Let's try a few examples.

Designing an algorithm

- In this class we will use the words “design”, “pseudocode”, and “algorithm” interchangeably.

Designing an algorithm

- In this class we will use the words "design", "pseudocode", and "algorithm" interchangeably.
- These are the steps to solve a problem.
- A design is like an outline or rough draft for your program.
- Figure out what you're going to do before you start doing it!
- We'll start with a non-computer example.

Designing an algorithm

- In this class we will use the words “design”, “pseudocode”, and “algorithm” interchangeably.
- These are the steps to solve a problem.
- A design is like an outline or rough draft for your program.
- Figure out what you’re going to do before you start doing it!
- We’ll start with a non-computer example.

Design: building a dog house

Let's say we want to build a dog house.

Design: building a dog house

Let's say we want to build a dog house. What steps do we need to take?

Design: building a dog house

Let's say we want to build a dog house. What steps do we need to take?

1. Decide on a location and size for the doghouse.
2. Get materials for the house.

Design: building a dog house

Let's say we want to build a dog house. What steps do we need to take?

1. Decide on a location and size for the doghouse.
2. Get materials for the house.
3. Cut a piece of wood for the floor.

Design: building a dog house

Let's say we want to build a dog house. What steps do we need to take?

1. Decide on a location and size for the doghouse.
2. Get materials for the house.
3. Cut a piece of wood for the floor.
4. Cut wood for the four walls.
5. Cut a door into one wall.

Design: building a dog house

Let's say we want to build a dog house. What steps do we need to take?

1. Decide on a location and size for the doghouse.
2. Get materials for the house.
3. Cut a piece of wood for the floor.
4. Cut wood for the four walls.
5. Cut a door into one wall.
6. Assemble walls.
7. Attach walls to the floor.

Design: building a dog house

Let's say we want to build a dog house. What steps do we need to take?

1. Decide on a location and size for the doghouse.
2. Get materials for the house.
3. Cut a piece of wood for the floor.
4. Cut wood for the four walls.
5. Cut a door into one wall.
6. Assemble walls.
7. Attach walls to the floor.
8. Make roof.
9. Attach roof to walls.

Design: building a dog house

Let's say we want to build a dog house. What steps do we need to take?

1. Decide on a location and size for the doghouse.
2. Get materials for the house.
3. Cut a piece of wood for the floor.
4. Cut wood for the four walls.
5. Cut a door into one wall.
6. Assemble walls.
7. Attach walls to the floor.
8. Make roof.
9. Attach roof to walls.
10. Paint the outside.

Design: building a dog house

Let's say we want to build a dog house. What steps do we need to take?

1. Decide on a location and size for the doghouse.
2. Get materials for the house.
3. Cut a piece of wood for the floor.
4. Cut wood for the four walls.
5. Cut a door into one wall.
6. Assemble walls.
7. Attach walls to the floor.
8. Make roof.
9. Attach roof to walls.
10. Paint the outside.

Notes on the design

- Steps are numbered in the order they should be performed.
  - If I try cutting the door after attaching the walls to the floor, it will be harder.

Notes on the design

- Steps are numbered in the order they should be performed.
  - If I try cutting the door after attaching the walls to the floor, it will be harder.
  - You'll number your steps for the first few designs in class.

Notes on the design

- Steps are numbered in the order they should be performed.
    - If I try cutting the door after attaching the walls to the floor, it will be harder.
    - You'll number your steps for the first few designs in class.
- Some steps could be further divided:
    - "Get materials": what materials? Where?

Notes on the design

- Steps are numbered in the order they should be performed.
  - If I try cutting the door after attaching the walls to the floor, it will be harder.
  - You'll number your steps for the first few designs in class.
- Some steps could be further divided:
  - "Get materials": what materials? Where?
  - "Make roof": cut at an angle, nail together, . . .

Notes on the design

- Steps are numbered in the order they should be performed.
    - If I try cutting the door after attaching the walls to the floor, it will be harder.
    - You'll number your steps for the first few designs in class.
- Some steps could be further divided:
    - "Get materials": what materials? Where?
    - "Make roof": cut at an angle, nail together, ...
- "Cut wood for the four walls": a repeated step.

Notes on the design

- Steps are numbered in the order they should be performed.
  - If I try cutting the door after attaching the walls to the floor, it will be harder.
  - You'll number your steps for the first few designs in class.
- Some steps could be further divided:
  - "Get materials": what materials? Where?
  - "Make roof": cut at an angle, nail together, . . .
- "Cut wood for the four walls": a repeated step.
- Could go into more detail: how big is a wall, the floor, etc.?

Notes on the design

- Steps are numbered in the order they should be performed.
    - If I try cutting the door after attaching the walls to the floor, it will be harder.
    - You'll number your steps for the first few designs in class.
- Some steps could be further divided:
    - "Get materials": what materials? Where?
    - "Make roof": cut at an angle, nail together, ...
- "Cut wood for the four walls": a repeated step.
- Could go into more detail: how big is a wall, the floor, etc.?
- What's the budget?

Dog house, refined

1. Decide on a location and size for the doghouse.
2. Get materials for the house.
    1. Get lumber.
    2. Get paint.
    3. Get nails.
3. Cut a piece of wood for the floor.
4. Repeat four times:
    1. Cut a piece of wood for a wall.
5. Cut a door into one wall.
6. Attach walls to the floor.
7. Make roof.
    1. Cut two pieces of wood.
    2. Join the pieces at a 90 degree angle.
    3. Nail the pieces together.
8. Attach roof to walls.
9. Paint the outside.

Dog house, refined

1. Decide on a location and size for the doghouse.
2. Get materials for the house.
    1. Get lumber.
    2. Get paint.
    3. Get nails.
3. Cut a piece of wood for the floor.
4. Repeat four times:
    1. Cut a piece of wood for a wall.
5. Cut a door into one wall.
6. Attach walls to the floor.
7. Make roof.
    1. Cut two pieces of wood.
    2. Join the pieces at a 90 degree angle.
    3. Nail the pieces together.
8. Attach roof to walls.
9. Paint the outside.

The first computers

- The first automatic computers were designed to solve one specific problem.

The first computers

- The first automatic computers were designed to solve one specific problem.
- The Antikythera mechanism was built around 100 BCE for calendar and astronomical calculations.

The first computers

- The fir ve one specific probler
- The Ar CE for calendar and ast

The Antikythera mechanism

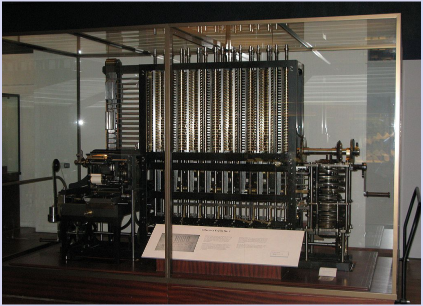
Image: user "Marsyas", Wikimedia Commons, 20 December 2005.

The first computers

- The first automatic computers were designed to solve one specific problem.
- The Antikythera mechanism was built around 100 BCE for calendar and astronomical calculations.
- Charles Babbage designed the difference engine, 1823–1842, to compute values of polynomials.
  - Never finished in his lifetime.
  - Finally built in 1991.

The first computers

- The fir ve one specific
  probler
- The Ar CE for calendar
  and ast
- Charles 23–1842, to
  comput
  - Ne
  - Fi



Babbage's difference engine, built 1991

Image: Allan J. Cronin, Wikipedia, March 2009.

The first computers

- The first automatic computers were designed to solve one specific problem.
- The Antikythera mechanism was built around 100 BCE for calendar and astronomical calculations.
- Charles Babbage designed the difference engine, 1823–1842, to compute values of polynomials.
  - Never finished in his lifetime.
  - Finally built in 1991.
  - And it worked!

The first computers

- The first automatic computers were designed to solve one specific problem.
- The Antikythera mechanism was built around 100 BCE for calendar and astronomical calculations.
- Charles Babbage designed the difference engine, 1823–1842, to compute values of polynomials.
  - Never finished in his lifetime.
  - Finally built in 1991.
  - And it worked!

Programming early computers

- Early computers were designed to solve one specific problem.
- Some could be reprogrammed by flipping switches or plugging in cables.

Programming early computers

- Early computers were designed to solve one specific problem.
- Some could be reprogrammed by flipping switches or plugging in cables.
  - Flip switches to enter a number into the “store”.
  - Connect cables from the store to the adder, multiplier, etc.

Programming early computers

- Early computers were designed to solve one specific problem.
- Some could be reprogrammed by flipping switches or plugging in cables.
    - Flip switches to enter a number into the "store".
    - Connect cables from the store to the adder, multiplier, etc.
    - Setting up the machine to solve a problem could take days.
        - *Even if* you already know which cables should go where.

Programming early computers

- Early computers were designed to solve one specific problem.
- Some could be reprogrammed by flipping switches or plugging in cables.
    - Flip switches to enter a number into the "store".
    - Connect cables from the store to the adder, multiplier, etc.
    - Setting up the machine to solve a problem could take days.
        - *Even if* you already know which cables should go where.
- But still, that was faster and more accurate than humans.

Programming early computers

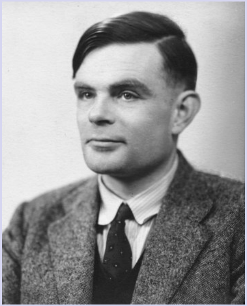
- Early computers were designed to solve one specific problem.
- Some could be reprogrammed by flipping switches or plugging in cables.
  - Flip switches to enter a number into the “store”.
  - Connect cables from the store to the adder, multiplier, etc.
  - Setting up the machine to solve a problem could take days.
    - *Even if* you already know which cables should go where.
- But still, that was faster and more accurate than humans.

Stored programs

- British mathematician Alan Turing described in 1936 a mathematical model of how machines can compute.

Stored programs

- British mathematician Alan Turing described in 1936 a mathematical model



Alan Turing, British computer scientist.

Image: National Portrait Gallery, London, 1951.

Stored programs

- British mathematician Alan Turing described in 1936 a mathematical model of how machines can compute.
    - A founder of modern computer science.

Stored programs

- British mathematician Alan Turing described in 1936 a mathematical model of how machines can compute.
  - A founder of modern computer science.
- He realized that you could make a **universal machine**
  - It would take as part of its input a description of the program to run.

Stored programs

- British mathematician Alan Turing described in 1936 a mathematical model of how machines can compute.
  - A founder of modern computer science.
- He realized that you could make a **universal machine**
  - It would take as part of its input a description of the program to run.
  - Programs become just another kind of data!
  - John von Neumann developed these ideas further in 1944.

Stored programs

- British mathematician Alan Turing described in 1936 a mathematical model
  - A
- He real **ne**
  - It e program to run.
  - Pr
  - Jo 1944.

John von Neumann, Hungarian-American computer scientist.

Image: Los Alamos National Lab.

Stored programs

- British mathematician Alan Turing described in 1936 a mathematical model of how machines can compute.
  - A founder of modern computer science.
- He realized that you could make a **universal machine**
  - It would take as part of its input a description of the program to run.
  - Programs become just another kind of data!
  - John von Neumann developed these ideas further in 1944.
- Turing later went on to develop the bombe to break WWII encryption.

Stored programs

- British mathematician Alan Turing described in 1936 a mathematical model
  - A
- He real ne
  - It e program to run.
  - Pr
  - Jo 1944.
- Turing WWII encryption.

Somebody set up us the bombe (reproduction).

Image: Antoine Taveneaux, Wikipedia, 18 June 2012.

Stored programs

- British mathematician Alan Turing described in 1936 a mathematical model of how machines can compute.
  - A founder of modern computer science.
- He realized that you could make a **universal machine**
  - It would take as part of its input a description of the program to run.
  - Programs become just another kind of data!
  - John von Neumann developed these ideas further in 1944.
- Turing later went on to develop the bombe to break WWII encryption.
  - Germany used the Enigma machine to encrypt wartime messages.
  - The bombe figured out which settings the Enigma used each day.

Stored programs

- British mathematician Alan Turing described in 1936 a mathematical model of how machines can compute.
    - A founder of modern computer science.
- He realized that you could make a **universal machine**
    - It would take as part of its input a description of the program to run.
    - Programs become just another kind of data!
    - John von Neumann developed these ideas further in 1944.
- Turing later went on to develop the bombe to break WWII encryption.
    - Germany used the Enigma machine to encrypt wartime messages.
    - The bombe figured out which settings the Enigma used each day.
    - 2014 film: *The Imitation Game*
    - "The Imitation Game" was his name for what we now call the "Turing test": how can we tell whether a computer is intelligent?

Stored programs

- British mathematician Alan Turing described in 1936 a mathematical model of how machines can compute.
    - A founder of modern computer science.
- He realized that you could make a **universal machine**
    - It would take as part of its input a description of the program to run.
    - Programs become just another kind of data!
    - John von Neumann developed these ideas further in 1944.
- Turing later went on to develop the bombe to break WWII encryption.
    - Germany used the Enigma machine to encrypt wartime messages.
    - The bombe figured out which settings the Enigma used each day.
    - 2014 film: *The Imitation Game*
    - "The Imitation Game" was his name for what we now call the "Turing test": how can we tell whether a computer is intelligent?
- A sad end.
    - In 1952, it came out that Turing was gay: illegal at the time.
    - He was convicted and sentenced to chemical castration (hormones).
    - Committed suicide in 1954.

Stored programs

- British mathematician Alan Turing described in 1936 a mathematical model of how machines can compute.
  - A founder of modern computer science.
- He realized that you could make a **universal machine**
  - It would take as part of its input a description of the program to run.
  - Programs become just another kind of data!
  - John von Neumann developed these ideas further in 1944.
- Turing later went on to develop the bombe to break WWII encryption.
  - Germany used the Enigma machine to encrypt wartime messages.
  - The bombe figured out which settings the Enigma used each day.
  - 2014 film: *The Imitation Game*
  - "The Imitation Game" was his name for what we now call the "Turing test": how can we tell whether a computer is intelligent?
- A sad end.
  - In 1952, it came out that Turing was gay: illegal at the time.
  - He was convicted and sentenced to chemical castration (hormones).
  - Committed suicide in 1954.

Parts of a modern computer

- RAM: the computer's "working memory".
    - "Random Access Memory"

Parts of a modern computer

- RAM: the computer's "working memory".
  - "Random Access Memory"
  - Made up of cells (words), each holding a number.
    - Represented in binary.

Parts of a modern computer

- RAM: the computer's "working memory".
  - "Random Access Memory"
  - Made up of cells (words), each holding a number.
    - Represented in binary.
  - Volatile: information is lost when the power goes out.
  - Fast (nanoseconds)
  - Relatively expensive.

Parts of a modern computer

- RAM: the computer's "working memory".
  - "Random Access Memory"
  - Made up of cells (words), each holding a number.
    - Represented in binary.
  - Volatile: information is lost when the power goes out.
  - Fast (nanoseconds)
  - Relatively expensive.
  - Von Neumann architecture: CPU reads instructions from RAM.

Parts of a modern computer

- RAM: the computer's "working memory".
  - "Random Access Memory"
  - Made up of cells (words), each holding a number.
    - Represented in binary.
  - Volatile: information is lost when the power goes out.
  - Fast (nanoseconds)
  - Relatively expensive.
  - Von Neumann architecture: CPU reads instructions from RAM.
- Secondary storage: hard drives, flash, DVD, ...
  - Persistent: data can be stored for years or decades.
  - Slow (microseconds to milliseconds: $< 1/1000$ the speed of RAM)
  - Relatively cheap.

Parts of a modern computer

- RAM: the computer's "working memory".
  - "Random Access Memory"
  - Made up of cells (words), each holding a number.
    - ★ Represented in binary.
  - Volatile: information is lost when the power goes out.
  - Fast (nanoseconds)
  - Relatively expensive.
  - Von Neumann architecture: CPU reads instructions from RAM.
- Secondary storage: hard drives, flash, DVD, ...
  - Persistent: data can be stored for years or decades.
  - Slow (microseconds to milliseconds: $< 1/1000$ the speed of RAM)
  - Relatively cheap.
  - Data must be transferred to RAM before the CPU can use it.

Parts of a modern computer

- RAM: the computer's "working memory".
  - "Random Access Memory"
  - Made up of cells (words), each holding a number.
    - Represented in binary.
  - Volatile: information is lost when the power goes out.
  - Fast (nanoseconds)
  - Relatively expensive.
  - Von Neumann architecture: CPU reads instructions from RAM.
- Secondary storage: hard drives, flash, DVD, . . .
  - Persistent: data can be stored for years or decades.
  - Slow (microseconds to milliseconds: $< 1/1000$ the speed of RAM)
  - Relatively cheap.
  - Data must be transferred to RAM before the CPU can use it.
- CPU: Central Processing Unit.
  - Reads instructions from RAM.
  - Executes (carries them out) in order.

Parts of a modern computer

- RAM: the computer's "working memory".
  - "Random Access Memory"
  - Made up of cells (words), each holding a number.
    - Represented in binary.
  - Volatile: information is lost when the power goes out.
  - Fast (nanoseconds)
  - Relatively expensive.
  - Von Neumann architecture: CPU reads instructions from RAM.
- Secondary storage: hard drives, flash, DVD, . . .
  - Persistent: data can be stored for years or decades.
  - Slow (microseconds to milliseconds: $< 1/1000$ the speed of RAM)
  - Relatively cheap.
  - Data must be transferred to RAM before the CPU can use it.
- CPU: Central Processing Unit.
  - Reads instructions from RAM.
  - Executes (carries them out) in order.
  - Simple instructions: add numbers, is-equal, skip to another instruction.

Parts of a modern computer

- RAM: the computer's "working memory".
  - "Random Access Memory"
  - Made up of cells (words), each holding a number.
    - Represented in binary.
  - Volatile: information is lost when the power goes out.
  - Fast (nanoseconds)
  - Relatively expensive.
  - Von Neumann architecture: CPU reads instructions from RAM.
- Secondary storage: hard drives, flash, DVD, . . .
  - Persistent: data can be stored for years or decades.
  - Slow (microseconds to milliseconds: $< 1/1000$ the speed of RAM)
  - Relatively cheap.
  - Data must be transferred to RAM before the CPU can use it.
- CPU: Central Processing Unit.
  - Reads instructions from RAM.
  - Executes (carries them out) in order.
  - Simple instructions: add numbers, is-equal, skip to another instruction.

Computer units

- RAM consists of bits: a circuit that stores a 1 or a 0.
- Bits are combined into **bytes**, usually 8 bits.

Computer units

- RAM consists of bits: a circuit that stores a 1 or a 0.
- Bits are combined into **bytes**, usually 8 bits.
  - **Binary numbers**: places are powers of two
    - 1, 2, 4, 8, 16, 32, 64, 128

Computer units

- RAM consists of bits: a circuit that stores a 1 or a 0.
- Bits are combined into **bytes**, usually 8 bits.
  - **Binary numbers**: places are powers of two
    - 1, 2, 4, 8, 16, 32, 64, 128
  - So 01001011

Computer units

- RAM consists of bits: a circuit that stores a 1 or a 0.
- Bits are combined into **bytes**, usually 8 bits.
    - **Binary numbers**: places are powers of two
        - 1, 2, 4, 8, 16, 32, 64, 128
    - So 0100101**1** = 1

Computer units

- RAM consists of bits: a circuit that stores a 1 or a 0.
- Bits are combined into **bytes**, usually 8 bits.
  - **Binary numbers**: places are powers of two
    - 1, 2, 4, 8, 16, 32, 64, 128
  - So 010010**1**1 $= 1 + 2$

Computer units

- RAM consists of bits: a circuit that stores a 1 or a 0.
- Bits are combined into **bytes**, usually 8 bits.
  - **Binary numbers**: places are powers of two
    - 1, 2, 4, 8, 16, 32, 64, 128
  - So 0100**1**011 $= 1 + 2 + 8$

Computer units

- RAM consists of bits: a circuit that stores a 1 or a 0.
- Bits are combined into **bytes**, usually 8 bits.
  - **Binary numbers**: places are powers of two
    - 1, 2, 4, 8, 16, 32, 64, 128
  - So 0**1**001011 $= 1 + 2 + 8 + 64$

Computer units

- RAM consists of bits: a circuit that stores a 1 or a 0.
- Bits are combined into **bytes**, usually 8 bits.
  - **Binary numbers**: places are powers of two
    - 1, 2, 4, 8, 16, 32, 64, 128
  - So $01001011 = 1 + 2 + 8 + 64 = 75$

Computer units

- RAM consists of bits: a circuit that stores a 1 or a 0.
- Bits are combined into **bytes**, usually 8 bits.
  - **Binary numbers**: places are powers of two
    - 1, 2, 4, 8, 16, 32, 64, 128
  - So $01001011 = 1 + 2 + 8 + 64 = 75$
  - (More about this in chapter 3).

Computer units

- RAM consists of bits: a circuit that stores a 1 or a 0.
- Bits are combined into **bytes**, usually 8 bits.
    - **Binary numbers**: places are powers of two
        - 1, 2, 4, 8, 16, 32, 64, 128
    - So $01001011 = 1 + 2 + 8 + 64 = 75$
    - (More about this in chapter 3).
    - One byte can represent a number from 0 to 255.
        - Or a single character in **ASCII** code.

Computer units

- RAM consists of bits: a circuit that stores a 1 or a 0.
- Bits are combined into **bytes**, usually 8 bits.
  - **Binary numbers**: places are powers of two
    - 1, 2, 4, 8, 16, 32, 64, 128
  - So $01001011 = 1 + 2 + 8 + 64 = 75$
  - (More about this in chapter 3).
  - One byte can represent a number from 0 to 255.
    - Or a single character in **ASCII** code.
- Kilobyte (kB): $2^{10} = 1024$ bytes (about a page of text)

Computer units

- RAM consists of bits: a circuit that stores a 1 or a 0.
- Bits are combined into **bytes**, usually 8 bits.
  - **Binary numbers**: places are powers of two
    - 1, 2, 4, 8, 16, 32, 64, 128
  - So $01001011 = 1 + 2 + 8 + 64 = 75$
  - (More about this in chapter 3).
  - One byte can represent a number from 0 to 255.
    - Or a single character in **ASCII** code.
- Kilobyte (kB): $2^{10} = 1024$ bytes (about a page of text)
- Megabyte (MB): $2^{20} \approx 1$ million bytes (1024 kB, a large book)
  - A song in MP3 format might take 3 or 4 MB.

Computer units

- RAM consists of bits: a circuit that stores a 1 or a 0.
- Bits are combined into **bytes**, usually 8 bits.
  - **Binary numbers**: places are powers of two
    - 1, 2, 4, 8, 16, 32, 64, 128
  - So $01001011 = 1 + 2 + 8 + 64 = 75$
  - (More about this in chapter 3).
  - One byte can represent a number from 0 to 255.
    - Or a single character in **ASCII** code.
- Kilobyte (kB): $2^{10} = 1024$ bytes (about a page of text)
- Megabyte (MB): $2^{20} \approx 1$ million bytes (1024 kB, a large book)
  - A song in MP3 format might take 3 or 4 MB.
- Gigabyte (GB): $2^{30} \approx 1$ billion bytes (1024 MB, a small library)
  - A DVD is about 4.7 GB.
  - A modern computer might have 16 GB of RAM.

Computer units

- RAM consists of bits: a circuit that stores a 1 or a 0.
- Bits are combined into **bytes**, usually 8 bits.
  - **Binary numbers**: places are powers of two
    - 1, 2, 4, 8, 16, 32, 64, 128
  - So $01001011 = 1 + 2 + 8 + 64 = 75$
  - (More about this in chapter 3).
  - One byte can represent a number from 0 to 255.
    - Or a single character in **ASCII** code.
- Kilobyte (kB): $2^{10} = 1024$ bytes (about a page of text)
- Megabyte (MB): $2^{20} \approx 1$ million bytes (1024 kB, a large book)
  - A song in MP3 format might take 3 or 4 MB.
- Gigabyte (GB): $2^{30} \approx 1$ billion bytes (1024 MB, a small library)
  - A DVD is about 4.7 GB.
  - A modern computer might have 16 GB of RAM.
- Terabyte (TB): $2^{40} \approx 1$ trillion bytes (1024 GB, a large library)
  - A modern hard drive might be 1 or 2 TB.

Calculating with computer units

- Let's say you have a 16 GB USB stick.
  - And a bunch of videos, 256 MB each.
  - How many videos can it hold?

Calculating with computer units

- Let's say you have a 16 GB USB stick.
  - And a bunch of videos, 256 MB each.
  - How many videos can it hold?
  - $1024 = 256 \times 4$

Calculating with computer units

- Let's say you have a 16 GB USB stick.
    - And a bunch of videos, 256 MB each.
    - How many videos can it hold?
    - $1024 = 256 \times 4$, so you can fit four videos in one GB.

Calculating with computer units

- Let's say you have a 16 GB USB stick.
  - And a bunch of videos, 256 MB each.
  - How many videos can it hold?
  - $1024 = 256 \times 4$, so you can fit four videos in one GB.
  - $4 \times 16 = 64$: 64 videos on the USB stick.

Calculating with computer units

- Let's say you have a 16 GB USB stick.
  - And a bunch of videos, 256 MB each.
  - How many videos can it hold?
  - $1024 = 256 \times 4$, so you can fit four videos in one GB.
  - $4 \times 16 = 64$: 64 videos on the USB stick.
- Beware!
  - Hard drive manufacturers use a different definition of kB, MB, etc!
  - They say that 1 kB is exactly 1000 bytes (not 1024).

Calculating with computer units

- Let's say you have a 16 GB USB stick.
  - And a bunch of videos, 256 MB each.
  - How many videos can it hold?
  - $1024 = 256 \times 4$, so you can fit four videos in one GB.
  - $4 \times 16 = 64$: 64 videos on the USB stick.
- Beware!
  - Hard drive manufacturers use a different definition of kB, MB, etc!
  - They say that 1 kB is exactly 1000 bytes (not 1024).
    - ⋆ And that 1 MB is exactly 1 million bytes, 1 GB exactly 1 billion...

Calculating with computer units

- Let's say you have a 16 GB USB stick.
  - And a bunch of videos, 256 MB each.
  - How many videos can it hold?
  - $1024 = 256 \times 4$, so you can fit four videos in one GB.
  - $4 \times 16 = 64$: 64 videos on the USB stick.
- Beware!
  - Hard drive manufacturers use a different definition of kB, MB, etc!
  - They say that 1 kB is exactly 1000 bytes (not 1024).
    - ⋆ And that 1 MB is exactly 1 million bytes, 1 GB exactly 1 billion...
  - When it gets to terabytes, that's a difference of 10%!

Calculating with computer units

- Let's say you have a 16 GB USB stick.
    - And a bunch of videos, 256 MB each.
    - How many videos can it hold?
    - $1024 = 256 \times 4$, so you can fit four videos in one GB.
    - $4 \times 16 = 64$: 64 videos on the USB stick.
- Beware!
    - Hard drive manufacturers use a different definition of kB, MB, etc!
    - They say that 1 kB is exactly 1000 bytes (not 1024).
        - And that 1 MB is exactly 1 million bytes, 1 GB exactly 1 billion. . .
    - When it gets to terabytes, that's a difference of 10%!
    - Sometimes you will see "KiB", "MiB", "GiB", "TiB":
        - "Kibibytes", "Mebibytes", "Gibibytes", "Tebibytes"
        - Unambiguously refer to the 1024 definition, not 1000.

Calculating with computer units

- Let's say you have a 16 GB USB stick.
  - And a bunch of videos, 256 MB each.
  - How many videos can it hold?
  - $1024 = 256 \times 4$, so you can fit four videos in one GB.
  - $4 \times 16 = 64$: 64 videos on the USB stick.
- Beware!
  - Hard drive manufacturers use a different definition of kB, MB, etc!
  - They say that 1 kB is exactly 1000 bytes (not 1024).
    - And that 1 MB is exactly 1 million bytes, 1 GB exactly 1 billion. . .
  - When it gets to terabytes, that's a difference of 10%!
  - Sometimes you will see "KiB", "MiB", "GiB", "TiB":
    - "Kibibytes", "Mebibytes", "Gibibytes", "Tebibytes"
    - Unambiguously refer to the 1024 definition, not 1000.

Programming languages

Computer programming is the process of translating an algorithm into step-by-step instructions a computer can understand.

Programming languages

Computer programming is the process of translating an algorithm into step-by-step instructions a computer can understand.

So what do these instructions look like?

- That depends. . .

Programming languages

Computer programming is the process of translating an algorithm into step-by-step instructions a computer can understand.

So what do these instructions look like?

- That depends. . .
- A **programming language** is a particular way of writing instructions to a computer.

Programming languages

Computer programming is the process of translating an algorithm into step-by-step instructions a computer can understand.
So what do these instructions look like?

- That depends. . .
- A **programming language** is a particular way of writing instructions to a computer.
- There are thousands of programming languages out there, dozens or hundreds of which are still in regular use.
  - A professional programmer usually knows several.
  - Then they can choose the right tool (language) for each job.
- In CS 115, we'll learn to write programs in **Python**, a high-level, interpreted programming language.

Syntax and semantics

In a given programming language:

- **Syntax** are the rules that say what programs look like

Syntax and semantics

In a given programming language:

- **Syntax** are the rules that say what programs look like:
  - Spelling.
  - Punctuation.
  - Order and combination of words (grammar).

Syntax and semantics

In a given programming language:

- **Syntax** are the rules that say what programs look like:
  - Spelling.
  - Punctuation.
  - Order and combination of words (grammar).
- **Semantics** are the rules that say what programs mean

Syntax and semantics

In a given programming language:

- **Syntax** are the rules that say what programs look like:
  - Spelling.
  - Punctuation.
  - Order and combination of words (grammar).
- **Semantics** are the rules that say what programs mean:
  - What does the computer do when it executes this statement?
  - When you combine these statements, what happens?

Syntax and semantics

In a given programming language:

- **Syntax** are the rules that say what programs look like:
  - Spelling.
  - Punctuation.
  - Order and combination of words (grammar).
- **Semantics** are the rules that say what programs mean:
  - What does the computer do when it executes this statement?
  - When you combine these statements, what happens?

Low-level languages

Low-level languages:

- **Machine code**: numbers treated as instructions by the CPU.

Low-level languages

Low-level languages:

- **Machine code**: numbers treated as instructions by the CPU.
  05 01

Low-level languages

Low-level languages:

- **Machine code**: numbers treated as instructions by the CPU.
  `05 01`
- **Assembly code**: human-readable way of writing machine code.

Low-level languages

Low-level languages:

- **Machine code**: numbers treated as instructions by the CPU.

  ```
  05 01
  ```

- **Assembly code**: human-readable way of writing machine code.

  ```
  add EAX, 1
  mov [ESP+4], EAX
  ```

Low-level languages

Low-level languages:

- **Machine code**: numbers treated as instructions by the CPU.
  ```
  05 01
  ```
- **Assembly code**: human-readable way of writing machine code.
  ```
  add EAX, 1
  mov [ESP+4], EAX
  ```
- A single instruction does very little.
- Different languages for Intel (32 and 64 bit), ARM, PowerPC, . . .

Low-level languages

Low-level languages:

- **Machine code**: numbers treated as instructions by the CPU.
  ```
  05 01
  ```
- **Assembly code**: human-readable way of writing machine code.
  ```
  add EAX, 1
  mov [ESP+4], EAX
  ```
- A single instruction does very little.
- Different languages for Intel (32 and 64 bit), ARM, PowerPC, . . .
- For many years, these were the only ways to write programs.

Low-level languages

Low-level languages:

- **Machine code**: numbers treated as instructions by the CPU.
  ```
  05 01
  ```
- **Assembly code**: human-readable way of writing machine code.
  ```
  add EAX, 1
  mov [ESP+4], EAX
  ```
- A single instruction does very little.
- Different languages for Intel (32 and 64 bit), ARM, PowerPC, . . .
- For many years, these were the only ways to write programs.
  - Difficult, verbose, error-prone, and machine-specific.

Low-level languages

Low-level languages:

- **Machine code**: numbers treated as instructions by the CPU.
  ```
  05 01
  ```
- **Assembly code**: human-readable way of writing machine code.
  ```
  add EAX, 1
  mov [ESP+4], EAX
  ```
- A single instruction does very little.
- Different languages for Intel (32 and 64 bit), ARM, PowerPC, ...
- For many years, these were the only ways to write programs.
  - Difficult, verbose, error-prone, and machine-specific.

High-level languages

Low-level languages have very simple instructions—you need lots of instructions to do anything useful.

High-level languages

Low-level languages have very simple instructions—you need lots of instructions to do anything useful. High-level languages like Python and C++ make things simpler: allow one statement to stand for many machine code instructions:

Assembly language:

```
mov EAX, EBP[-2]
mov EBX, EBP[-4]
add EBX, 100
mul EAX, EBX
div EAX, 100
mov EBP[2], EAX
```

High-level languages

Low-level languages have very simple instructions—you need lots of instructions to do anything useful. High-level languages like Python and C++ make things simpler: allow one statement to stand for many machine code instructions:

Assembly language:

```
mov EAX, EBP[-2]
mov EBX, EBP[-4]
add EBX, 100
mul EAX, EBX
div EAX, 100
mov EBP[2], EAX
```

High-level language:

```
total = price * (tax + 100) / 100
```

High-level languages

Low-level languages have very simple instructions—you need lots of instructions to do anything useful. High-level languages like Python and C++ make things simpler: allow one statement to stand for many machine code instructions:

Assembly language:

```
load r1, -2[sp]
mov r2, -4[sp]
load r3, 100
add r2, r3, r2
mul r1, r2, r4
div r4, r3, r5
store sp[2], r1
```

High-level language:

```
total = price * (tax + 100) / 100
```

And you can translate it into different machine code instructions for another processor.

High-level languages

Low-level languages have very simple instructions—you need lots of instructions to do anything useful. High-level languages like Python and C++ make things simpler: allow one statement to stand for many machine code instructions:

Assembly language:

```
load r1, -2[sp]
mov r2, -4[sp]
load r3, 100
add r2, r3, r2
mul r1, r2, r4
div r4, r3, r5
store sp[2], r1
```

High-level language:

```
total = price * (tax + 100) / 100
```

And you can translate it into different machine code instructions for another processor.

Interpreters and compilers

Underneath, the computer still understands only machine code. So if we write in a high-level language, we have to have a way to translate that language into machine code.

Interpreters and compilers

Underneath, the computer still understands only machine code. So if we write in a high-level language, we have to have a way to translate that language into machine code.

There are two general ways to do this: interpreters and compilers.

Interpreters and compilers

Underneath, the computer still understands only machine code. So if we write in a high-level language, we have to have a way to translate that language into machine code.

There are two general ways to do this: interpreters and compilers.

- **Interpreter**: deciphers a language and **executes** instructions in order.

Interpreters and compilers

Underneath, the computer still understands only machine code. So if we write in a high-level language, we have to have a way to translate that language into machine code.

There are two general ways to do this: interpreters and compilers.

- **Interpreter**: deciphers a language and **executes** instructions in order.
  - + Easy to change your program: edit source, run again.
  - − Must decode the program each time: slow.
  - − Users need a copy of the interpreter.

Interpreters and compilers

Underneath, the computer still understands only machine code. So if we write in a high-level language, we have to have a way to translate that language into machine code.

There are two general ways to do this: interpreters and compilers.

- **Interpreter**: deciphers a language and **executes** instructions in order.
  - + Easy to change your program: edit source, run again.
  - – Must decode the program each time: slow.
  - – Users need a copy of the interpreter.
  - Examples: **Python**, JavaScript, Perl.

Interpreters and compilers

Underneath, the computer still understands only machine code. So if we write in a high-level language, we have to have a way to translate that language into machine code.

There are two general ways to do this: interpreters and compilers.

- **Interpreter**: deciphers a language and **executes** instructions in order.
  - + Easy to change your program: edit source, run again.
  - – Must decode the program each time: slow.
  - – Users need a copy of the interpreter.
  - ▸ Examples: **Python**, JavaScript, Perl.
- **Compiler**: deciphers a language and **translates** it to machine code.
  - ▸ (without executing it!)

Interpreters and compilers

Underneath, the computer still understands only machine code. So if we write in a high-level language, we have to have a way to translate that language into machine code.

There are two general ways to do this: interpreters and compilers.

- **Interpreter**: deciphers a language and **executes** instructions in order.
  - + Easy to change your program: edit source, run again.
  - – Must decode the program each time: slow.
  - – Users need a copy of the interpreter.
  - ▸ Examples: **Python**, JavaScript, Perl.
- **Compiler**: deciphers a language and **translates** it to machine code.
  - ▸ (without executing it!)
  - – Changing a program requires an additional step (compiling).
  - + Compile once, execute many times: runs faster.
  - + Run directly by the operating system—no translator needed.

Interpreters and compilers

Underneath, the computer still understands only machine code. So if we write in a high-level language, we have to have a way to translate that language into machine code.

There are two general ways to do this: interpreters and compilers.

- **Interpreter**: deciphers a language and **executes** instructions in order.
  - + Easy to change your program: edit source, run again.
  - – Must decode the program each time: slow.
  - – Users need a copy of the interpreter.
  - ▸ Examples: **Python**, JavaScript, Perl.
- **Compiler**: deciphers a language and **translates** it to machine code.
  - ▸ (without executing it!)
  - – Changing a program requires an additional step (compiling).
  - + Compile once, execute many times: runs faster.
  - + Run directly by the operating system—no translator needed.
  - ▸ Examples: C++, FORTRAN, Haskell.

Interpreters and compilers

Underneath, the computer still understands only machine code. So if we write in a high-level language, we have to have a way to translate that language into machine code.

There are two general ways to do this: interpreters and compilers.

- **Interpreter**: deciphers a language and **executes** instructions in order.
  - + Easy to change your program: edit source, run again.
  - – Must decode the program each time: slow.
  - – Users need a copy of the interpreter.
  - ▸ Examples: **Python**, JavaScript, Perl.
- **Compiler**: deciphers a language and **translates** it to machine code.
  - ▸ (without executing it!)
  - – Changing a program requires an additional step (compiling).
  - + Compile once, execute many times: runs faster.
  - + Run directly by the operating system—no translator needed.
  - ▸ Examples: C++, FORTRAN, Haskell.

Some languages combine features of both: Java is compiled into an intermediate **byte code**

Interpreters and compilers

Underneath, the computer still understands only machine code. So if we write in a high-level language, we have to have a way to translate that language into machine code.

There are two general ways to do this: interpreters and compilers.

- **Interpreter**: deciphers a language and **executes** instructions in order.
  - + Easy to change your program: edit source, run again.
  - – Must decode the program each time: slow.
  - – Users need a copy of the interpreter.
  - Examples: **Python**, JavaScript, Perl.
- **Compiler**: deciphers a language and **translates** it to machine code.
  - (without executing it!)
  - – Changing a program requires an additional step (compiling).
  - + Compile once, execute many times: runs faster.
  - + Run directly by the operating system—no translator needed.
  - Examples: C++, FORTRAN, Haskell.

Some languages combine features of both: Java is compiled into an intermediate **byte code**, which is then interpreted.

Interpreters and compilers

Underneath, the computer still understands only machine code. So if we write in a high-level language, we have to have a way to translate that language into machine code.

There are two general ways to do this: interpreters and compilers.

- **Interpreter**: deciphers a language and **executes** instructions in order.
  - + Easy to change your program: edit source, run again.
  - – Must decode the program each time: slow.
  - – Users need a copy of the interpreter.
  - Examples: **Python**, JavaScript, Perl.
- **Compiler**: deciphers a language and **translates** it to machine code.
  - (without executing it!)
  - – Changing a program requires an additional step (compiling).
  - + Compile once, execute many times: runs faster.
  - + Run directly by the operating system—no translator needed.
  - Examples: C++, FORTRAN, Haskell.

Some languages combine features of both: Java is compiled into an intermediate **byte code**, which is then interpreted.

An analogy

Suppose you want to make baklava (a kind of dessert pastry).

Image: Robert Kindermann, Wikipedia, 29 July 2006.

An analogy

Suppose you want to make baklava (a kind of dessert pastry). You have a recipe

Image: Robert Kindermann, Wikipedia, 29 July 2006.

An analogy

Suppose you want to make baklava (a kind of dessert pastry). You have a recipe, but it's in Greek, which you don't speak.

Μπακλαβάς

An analogy

Suppose you want to make baklava (a kind of dessert pastry). You have a recipe, but it's in Greek, which you don't speak. You have a friend who speaks Greek and English, but doesn't know how to bake.

Μπακλαβάς

An analogy

Suppose you want to make baklava (a kind of dessert pastry). You have a recipe, but it's in Greek, which you don't speak. You have a friend who speaks Greek and English, but doesn't know how to bake. What to do?

Μπακλαβάς

An analogy

Suppose you want to make baklava (a kind of dessert pastry). You have a recipe, but it's in Greek, which you don't speak. You have a friend who speaks Greek and English, but doesn't know how to bake. What to do? Two options:

1. Have your friend stand with you in the kitchen, telling you each instruction in order

An analogy

Suppose you want to make baklava (a kind of dessert pastry). You have a recipe, but it's in Greek, which you don't speak. You have a friend who speaks Greek and English, but doesn't know how to bake. What to do? Two options:

1. Have your friend stand with you in the kitchen, telling you each instruction in order—an **interpreter**.

An analogy

Suppose you want to make baklava (a kind of dessert pastry). You have a recipe, but it's in Greek, which you don't speak. You have a friend who speaks Greek and English, but doesn't know how to bake. What to do?

Two options:

1. Have your friend stand with you in the kitchen, telling you each instruction in order—an **interpreter**.
2. Give your friend the recipe and have them translate it into an English recipe and write it down

An analogy

Suppose you want to make baklava (a kind of dessert pastry). You have a recipe, but it's in Greek, which you don't speak. You have a friend who speaks Greek and English, but doesn't know how to bake. What to do? Two options:

1. Have your friend stand with you in the kitchen, telling you each instruction in order—an **interpreter**.
2. Give your friend the recipe and have them translate it into an English recipe and write it down—a **compiler**.

An analogy

Suppose you want to make baklava (a kind of dessert pastry). You have a recipe, but it's in Greek, which you don't speak. You have a friend who speaks Greek and English, but doesn't know how to bake. What to do? Two options:

1. Have your friend stand with you in the kitchen, telling you each instruction in order—an **interpreter**.
2. Give your friend the recipe and have them translate it into an English recipe and write it down—a **compiler**.

You can get started more quickly with the interpretation method, but you need your friend in the kitchen every single time.

An analogy

Suppose you want to make baklava (a kind of dessert pastry). You have a recipe, but it's in Greek, which you don't speak. You have a friend who speaks Greek and English, but doesn't know how to bake. What to do? Two options:

1. Have your friend stand with you in the kitchen, telling you each instruction in order—an **interpreter**.
2. Give your friend the recipe and have them translate it into an English recipe and write it down—a **compiler**.

You can get started more quickly with the interpretation method, but you need your friend in the kitchen every single time.

Programming environment and tools

What do you need to write programs in Python?

- An interpreter to translate and execute your program.

Programming environment and tools

What do you need to write programs in Python?

- An interpreter to translate and execute your program.
- A **text editor** for writing and changing source code.

Programming environment and tools

What do you need to write programs in Python?

- An interpreter to translate and execute your program.
- A **text editor** for writing and changing source code.
  - Notepad is common, but not really suited for programming.
  - More advanced editors can:
    - Automatically indent code.
    - Color code to clarify its meaning.
    - Jump from variable name to definition.
    - Much more.

Programming environment and tools

What do you need to write programs in Python?

- An interpreter to translate and execute your program.
- A **text editor** for writing and changing source code.
  - Notepad is common, but not really suited for programming.
  - More advanced editors can:
    - Automatically indent code.
    - Color code to clarify its meaning.
    - Jump from variable name to definition.
    - Much more.
- A **debugger** to help find and repair bugs.
  - Pause execution at a certain line.
  - Step through code line-by-line.
  - Inspect and change variables and memory.

Programming environment and tools

What do you need to write programs in Python?

- An interpreter to translate and execute your program.
- A **text editor** for writing and changing source code.
  - Notepad is common, but not really suited for programming.
  - More advanced editors can:
    - Automatically indent code.
    - Color code to clarify its meaning.
    - Jump from variable name to definition.
    - Much more.
- A **debugger** to help find and repair bugs.
  - Pause execution at a certain line.
  - Step through code line-by-line.
  - Inspect and change variables and memory.
- These are just some of the tools used by professional programmers.

Programming environment and tools

What do you need to write programs in Python?

- An interpreter to translate and execute your program.
- A **text editor** for writing and changing source code.
    - Notepad is common, but not really suited for programming.
    - More advanced editors can:
        - Automatically indent code.
        - Color code to clarify its meaning.
        - Jump from variable name to definition.
        - Much more.
- A **debugger** to help find and repair bugs.
    - Pause execution at a certain line.
    - Step through code line-by-line.
    - Inspect and change variables and memory.
- These are just some of the tools used by professional programmers.

Integrated development environments

- Many programmers build their toolkits by selecting their favorite tool for each job

Integrated development environments

- Many programmers build their toolkits by selecting their favorite tool for each job: interpreter, editor, debugger, etc.

Integrated development environments

- Many programmers build their toolkits by selecting their favorite tool for each job: interpreter, editor, debugger, etc.
- An **integrated development environment** (IDE): combines several programming tools into one cohesive program.

Integrated development environments

- Many programmers build their toolkits by selecting their favorite tool for each job: interpreter, editor, debugger, etc.
- An **integrated development environment** (IDE): combines several programming tools into one cohesive program.
- Some IDEs for Python:
  - IDLE: comes with Python.
  - WingIDE: recommended for this class.
- Lab 1 will introduce WingIDE.

Integrated development environments

- Many programmers build their toolkits by selecting their favorite tool for each job: interpreter, editor, debugger, etc.
- An **integrated development environment** (IDE): combines several programming tools into one cohesive program.
- Some IDEs for Python:
  - IDLE: comes with Python.
  - WingIDE: recommended for this class.
- Lab 1 will introduce WingIDE.
- Debugging and other topics in a few weeks.

Integrated development environments

- Many programmers build their toolkits by selecting their favorite tool for each job: interpreter, editor, debugger, etc.
- An **integrated development environment** (IDE): combines several programming tools into one cohesive program.
- Some IDEs for Python:
  - IDLE: comes with Python.
  - WingIDE: recommended for this class.
- Lab 1 will introduce WingIDE.
- Debugging and other topics in a few weeks.

The End

- Don't hesitate to email or visit office hours.
- Next lecture:
  - Installing and using Python and WingIDE.
  - A first Python program.
  - Documentation and comments.
  - Programming errors and debugging.
  - Variables, identifiers, and assignment.
  - Arithmetic.
- What questions do you have?

The End

- Don't hesitate to email or visit office hours.
- Next lecture:
  - Installing and using Python and WingIDE.
  - A first Python program.
  - Documentation and comments.
  - Programming errors and debugging.
  - Variables, identifiers, and assignment.
  - Arithmetic.
- What questions do you have?

How to do a design in CS 115

- Use a plain text editor, *not* a word processor.
  - The editor in IDLE or WingIDE works.
  - Notepad works.
  - Mac TextEdit: Format → Make Plain Text.

How to do a design in CS 115

- Use a plain text editor, *not* a word processor.
  - The editor in IDLE or WingIDE works.
  - Notepad works.
  - Mac TextEdit: Format → Make Plain Text.
- State the purpose of the program up top.
  - Followed by your name, section, email, assignment number.

How to do a design in CS 115

- Use a plain text editor, *not* a word processor.
  - The editor in IDLE or WingIDE works.
  - Notepad works.
  - Mac TextEdit: Format → Make Plain Text.
- State the purpose of the program up top.
  - Followed by your name, section, email, assignment number.
- One step per line.
  - Start the line with a "#" symbol (we'll see why next time).
  - Indent and number substeps and repeated steps.
  - Can number them 7.1, 7.2; or a, b, c: just make it clear.

How to do a design in CS 115

- Use a plain text editor, *not* a word processor.
  - The editor in IDLE or WingIDE works.
  - Notepad works.
  - Mac TextEdit: Format → Make Plain Text.
- State the purpose of the program up top.
  - Followed by your name, section, email, assignment number.
- One step per line.
  - Start the line with a "#" symbol (we'll see why next time).
  - Indent and number substeps and repeated steps.
  - Can number them 7.1, 7.2; or a, b, c: just make it clear.
- Hint: wait until the very end to number the steps.
  - That way there is less to change if you have to rearrange your design.

How to do a design in CS 115

- Use a plain text editor, *not* a word processor.
  - The editor in IDLE or WingIDE works.
  - Notepad works.
  - Mac TextEdit: Format → Make Plain Text.
- State the purpose of the program up top.
  - Followed by your name, section, email, assignment number.
- One step per line.
  - Start the line with a "#" symbol (we'll see why next time).
  - Indent and number substeps and repeated steps.
  - Can number them 7.1, 7.2; or a, b, c: just make it clear.
- Hint: wait until the very end to number the steps.
  - That way there is less to change if you have to rearrange your design.
- Give your file a name ending in `.py` (Python code)
  - Why? The design will be the basis for your implementation.
  - You'll write code for each step of the design.
    - Before long, you'll have a working program.

How to do a design in CS 115

- Use a plain text editor, *not* a word processor.
  - The editor in IDLE or WingIDE works.
  - Notepad works.
  - Mac TextEdit: Format → Make Plain Text.
- State the purpose of the program up top.
  - Followed by your name, section, email, assignment number.
- One step per line.
  - Start the line with a "#" symbol (we'll see why next time).
  - Indent and number substeps and repeated steps.
  - Can number them 7.1, 7.2; or a, b, c: just make it clear.
- Hint: wait until the very end to number the steps.
  - That way there is less to change if you have to rearrange your design.
- Give your file a name ending in `.py` (Python code)
  - Why? The design will be the basis for your implementation.
  - You'll write code for each step of the design.
    - Before long, you'll have a working program.

Example program design

```
# Purpose:   Ask for the user's name and greet them.
# Author:    J. Random Hacker, section 1,
#            random.hacker@uky.edu
# Assignment: Lab 42
# Main program:
# 1. Input the user's name from the keyboard
# 2. Output the word hello, followed by the user's name.
```

Turned into code

We'll see more about how this code works next time.

```
# Purpose:  Ask for the user's name and greet them.
# Author:   J. Random Hacker, section 1,
#           random.hacker@uky.edu
# Assignment: Lab 42
# Main program:
def main():
# 1. Input the user's name from the keyboard
    name = input("What is your name? ")
# 2. Output the word hello, followed by the user's name.
    print("Hello ", name)
main()
```