What is computer science?

- Lots of definitions – look up on Google
  - Most agree it differs from other sciences – no discovering what computers are, what they do, …
  - Our text offers a very concise definition: "Computer science is the study of algorithms."
- Okay, but what is an algorithm?
  - Simply: a *step-by-step* procedure to solve a problem
- So computer science is about solving problems
  - By using a computer of course
  - And therefore, it clearly is an *engineering* science

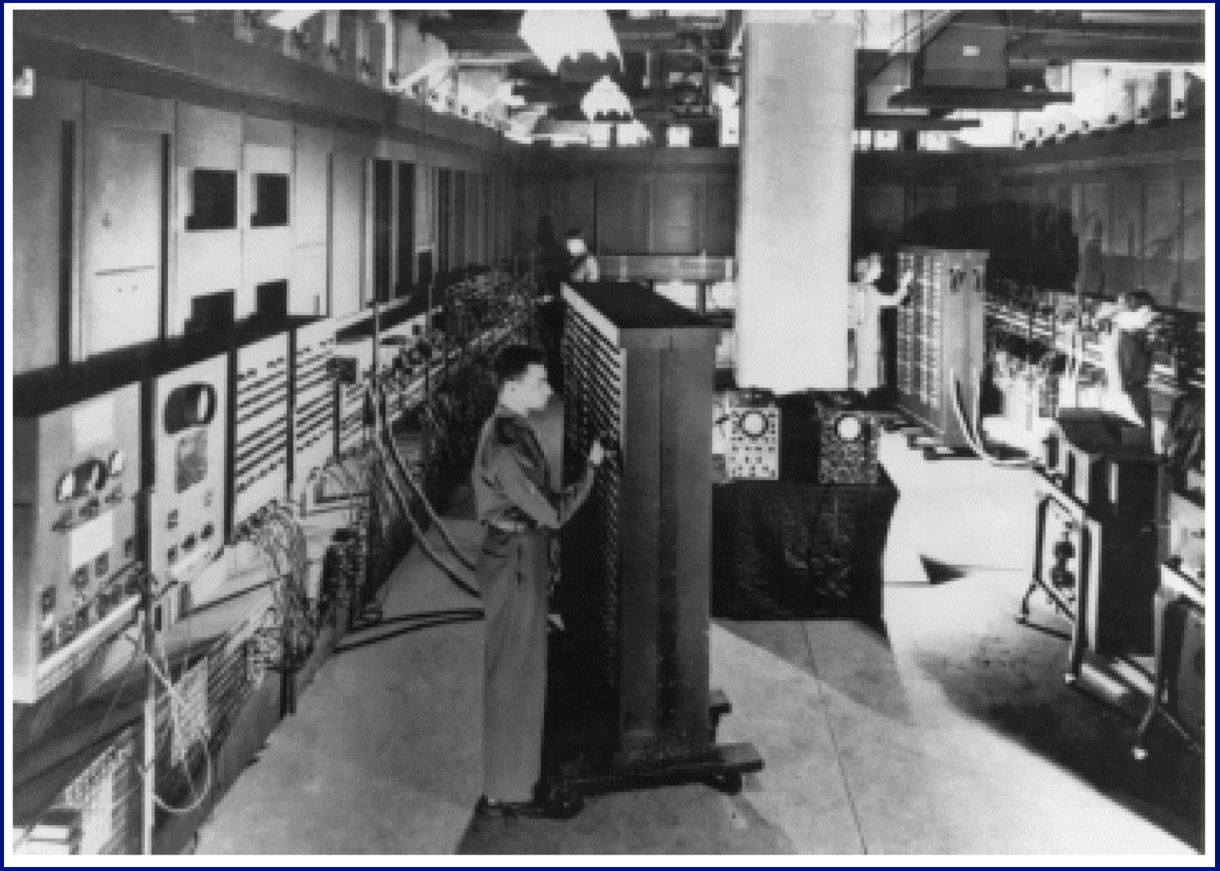
Problem-Solving Strategizing

- Helps to think about a problem at different scales
  - Big picture first – devise a general, overall strategy
  - Then progressively refine the overall solution by applying tactics and tools
  - Overall approach in computer science is known as "top-down programming by stepwise refinement"
- Best strategies, tactics and tools vary by problem
  - Idea: learn techniques applicable to many situations
- But first learn about our basic tools – computers

What is a computer?

- Webster: "one that computes"
  - Compute: "to determine esp. by mathematical means"
  - Abacus?
  - Slide rule?
- Person?
  - Actually a 1940s job title!
    - Ballistics project for U.S. War Dept. – computed artillery trajectories by desk calculator – up to 30-40 hours each
  - Led to the first electronic computer – the ENIAC

The ENIAC – electronic numerical integrator and computer – 1945



- 100 feet long, by 10 feet high, by 3 feet deep
- 30 tons!
- 17,468 vacuum tubes, 70,000 resistors, and 6,000 switches
- Trajectories computed in 30 seconds instead of 40 hours

Electronic computer hardware

- Central processing unit – CPU
  - Controls the other components, performs arithmetic, directs the flow of all data
- Main memory – a.k.a. RAM ("random access")
  - Fastest access, but short term – power must be on
  - States are binary – e.g., electronic pulse up or down
  - Also ROM ("read-only") – mostly for starting up
- Secondary storage – disks, CDs, tapes, …
  - Long-term memory – usually magnetic, so no power
- Input/output – I/O – keyboard, mouse, monitor, …

Hardware evolution

- Vacuum tubes phased out long ago
  - Replaced by transistors – faster, smaller, cheaper
  - Then by integrated circuits – “chips”
    - Millions of transistors – keep getting faster, smaller, cheaper
- I/O and storage improvements too
  - Direct wiring → IBM cards → keyboard → wireless
  - Line printer → dot-matrix → laser/color & more
  - Disk drums & 9-track tapes → multi-gigabyte → multi-*terabyte* (>1,000 gig) drives …

Today: “Personal” Computers

PC hardware – schematic

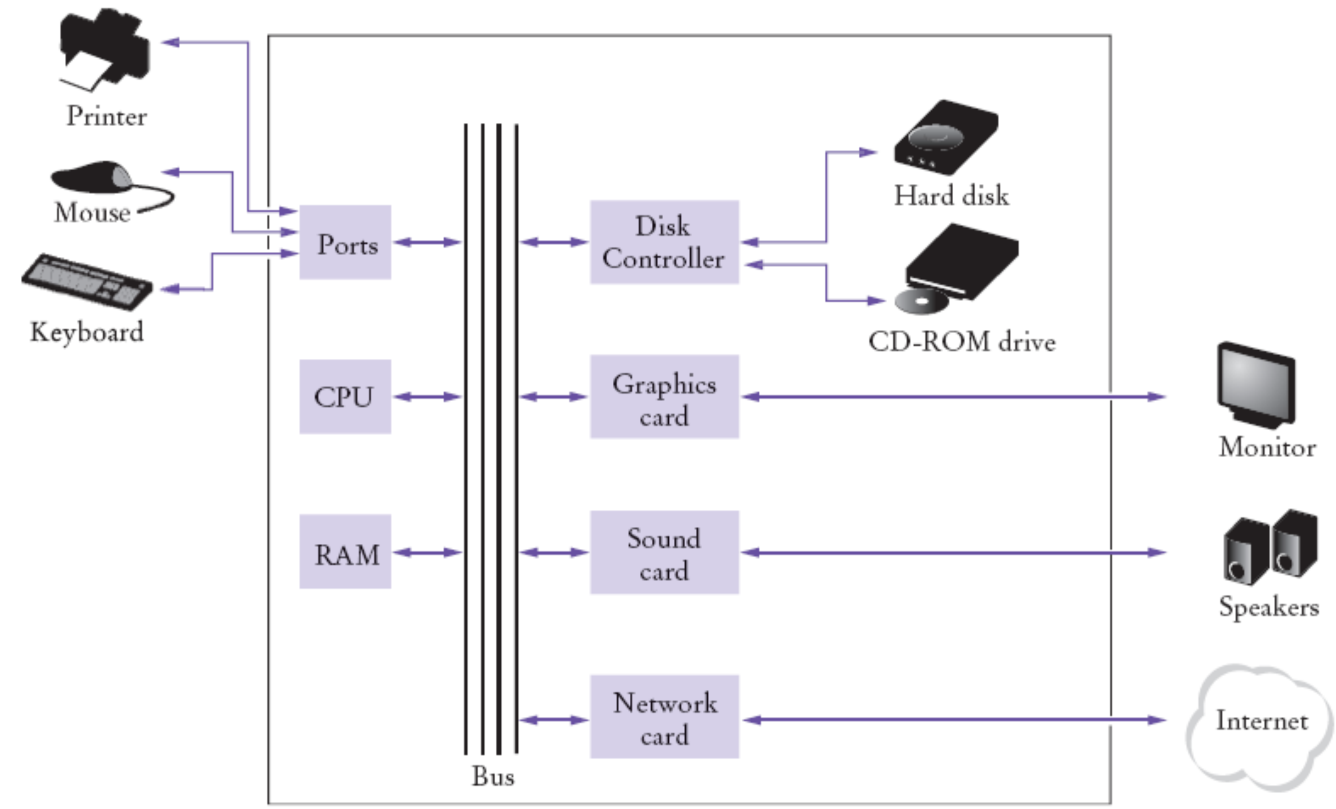


Figure 5 Schematic Diagram of a Computer

What is programming?

- Basically: instructing a computer what to do
- Programs – a.k.a. "Software"
  - Includes operating system, utilities, applications, …
  - Computer just sits there until instructions fed to CPU
- Machine language – basic CPU instructions
  - Completely numeric – i.e., computer "readable"
    - e.g., `43065932752`, might mean add (operation `43`) value at memory address `065` to value at address `932` and store result at address `752`
    - But in binary form, of course – `1001101`…
  - Specific to particular computer types – not portable

Programming languages

- Assembly language – 1st real advance
  - Human-readable instructions – translated to machine language by assembler programs
    - e.g., `ADD  X  Y  T`
    - Symbolic names represent operations and memory addresses
  - Very basic – lots of instructions to do simple things
  - Still processor-specific
- High-level languages – much bigger advance
  - Easier to write/read: `result = (first + second)`
  - Translated to assembly language (usually) by compiler programs
    - Same code works on many types of processors

High-level language paradigms

- Procedural languages – focus on *functions*
  - Fortran (by IBM, 1957) – first high level language
    - Easy to learn – spawned thousands of new programmers
  - C, Pascal, others – developed through 1970s
    - Even easier to learn/use – ever more programmers into 1990s
- Object-oriented languages – focus on *objects*
  - C++ (early 1980s), …, Java (1996)
  - Idea is to build objects – then let them perform tasks
- Multi-paradigm languages – combined features
  - e.g., Python (1991… and still evolving)

python™ ~1990…2017…

- Derived from ABC – a language designed for learning how to program
  - By Guido van Rossum (an ABC designer) – to be a more general purpose language than ABC

1995 photo

- Open source since version 1.0 (1991)
  - So it is free!
  - Huge community of volunteer developers
  - Guido still the BDFL (Benevolent Dictator for Life)

2014 photo

- Lots of handy modules ready to use
  http://docs.python.org/3.6/

Btw, not named for a snake

The Python interpreter

- A program that performs three steps over and over and …until `exit()`

1) It reads Python statements
   - From standard input (a.k.a. `stdin`; usually keyboard)
   - Or from a text file (usually named `.py`)
2) It executes Python commands
3) It prints results of commands if there are any

Try some arithmetic with it!

Numbers are objects to Python

- Each object *type* has: data and related operations
- 2 basic number types and one derived type
  - Integers (`5`, `-72`) – add, subtract, multiply, …
  - Floating point numbers (`0.005`, `-7.2`) – operations similar but *not exactly the same as integer* operations
  - Complex numbers – have two floating point parts, but operations are specific to complex numbers
- Expect many non-number object types later
  - But they still will have data and related operations

Arithmetic summary

- Operators:

  `+`, `-`, `*`, `/` add, subtract, multiply, (ordinary) divide

  `%` modulus operator – remainder

  `( )` means whatever is inside is evaluated first

- Special Python division operator for integers:

  `//` result is truncated: `7 // 2` → `3` (not 3.5)

- Precedence rules so far (will expand):

  1. `( )`
  2. `*`, `/`, `%`, `//`
  3. `+`, `-`
  4. `=`

Assigning names to objects

- Requires the assignment operator: `=`

  `x = 14` # Now `x` refers to integer object `14`
- Object names are actually references
  - Like "pointers" to objects
  - Can have multiple references to the same object

    `y = x` # Now `x` and `y` refer to the same object
- Dynamic typing is a key Python feature
  - Means any legal name can point to any type – even different types at different times

    `x = 1.2` # Now `x` refers to floating point (`y` still refers to `14`)

Names of objects (a.k.a. variables)

- 3 simple rules for choosing names:
  - Letters, digits, and _ (underscores) only
  - May not begin with a digit
  - No Python keywords (see Table 1.1 on p. 22)
- Also some advisories/conventions to follow:
  - Choose brief, but *meaningful* names
  - Avoid names of common Python modules, types, etc.
  - Most programmers prefer lower case – use "camel case" or underscore to separate words (`aCat`, or `a_cat`)
- All above apply to functions, modules, & types too

Abstraction

- Text def: "*a concept or idea not associated with any specific instance.*"
- A function, for example, is a kind of procedural abstraction

  `25` → Square Root Function → `5`

  - What goes on inside the function?
  - Doesn't matter, as long as it works!
- A Turtle, for example is a kind of data abstraction – and it has some functions too

Try it!

Using functions/methods

- Formally, to use (a.k.a. invoke) a function:

  `functionName(`*list of arguments*`)`

  - Effect – transfers control to the function named; may "pass" data to the function via the list of arguments
  - When function completes – control returns to the point in the program where the function was called
    - May also return a result – depends on the function definition
- Need "." (dot operator) if the function is defined in a module or if it is a class method
  - Then full syntax is `moduleName.functionName(…)` or `objectReference.methodName(…)`

Defining your own function

- Formally:
  `def` *name* `(` *list of parameters* `):`
  `#` *a block of statements here (all indented)*

  - `def` – mandatory keyword defines a function
  - *name* – any legal Python identifier
  - `( ):` – mandatory set of parentheses and colon
  - parameters – object names
    - Local references to objects that are passed into the function
    - May be an empty list
- By the way, # denotes a comment – actual statements would not be preceded by the comment character

A function to draw a square

- Part of listing 1.2 from the text (p. 30)

```
def drawSquare(myTurtle,sideLength):
    myTurtle.forward(sideLength)
    myTurtle.right(90)    # side 1
    myTurtle.forward(sideLength)
    myTurtle.right(90)    # side 2
    …
```

- Then to invoke it for drawing a square that has 20 pixels on each side using a turtle named `t`:

```
>>> drawSquare(t, 20)
```

- What might happen if `drawSquare(20, t)`?

Importing from a module

- Imagine the `drawSquare` function is in a file called `ds.py` – then two basic choices to use:
  1. Import whole module, and specify module to use
     ```
     >>> import ds
     >>> ds.drawSquare(t, 20)
     ```
  2. Import part(s) of module, then just use the part(s)
     ```
     >>> from ds import drawSquare
     ```
     # or [`from ds import *`] – gets all parts
     ```
     >>> drawSquare(t, 20)
     ```
  – Of course, Python must know where `ds.py` is
    - Store it in current directory or along `sys.path`
- Or in IDLE: *File*→*Open* – no need to import

Repetition with a `for` loop

- `for` *ref* `in` *a list*`:`
  `#` *block – ref refers to current object in list*
  - `for`, `in`, `:` – mandatory parts
  - *ref* – a name for referring to objects in the list
- The `range` function provides a handy list
  - Simplest: `range(n)` – a list with `n` items `[0,1,…n-1]`
  - Or: `range(start,stop)` – `[start, … stop-1]`
  - Or: `range(start, stop, step)` – `step` instead of `1`
  - `for i in range(1, 11, 4): #` iterates three times
    `print(i)      #` `i` is `1`, then `5`, then `9`

Simpler drawing by repetition

- Listing 1.3 from the text (p. 34)

```
def drawSquare2(myTurtle, sideLength):
    for i in range(4):
        myTurtle.forward(sideLength)
        myTurtle.right(90)
```

- Small variation draws a spiral (Listing 1.4)

```
def drawSpiral(myTurtle, maxSide):
    for sideLength in range(1, maxSide+1, 5):
        myTurtle.forward(sideLength)
        myTurtle.right(90)
```

More drawing abstraction

- Contrast – a triangle vs. a square (Listing 1.5)

```
def drawTriangle(myTurtle, sideLength):
    for i in range(3):  # draw 3 sides, not 4
        myTurtle.forward(sideLength)
        myTurtle.right(120)  # 120° × 3, not 90° × 4
```

- Hmm…any regular polygon? (Listing 1.6, p. 38)

```
def drawPolygon(myTurtle,sideLength,numSides):
    turnAngle = 360 / numSides
    for i in range(numSides):
        myTurtle.forward(sideLength)
        myTurtle.right(turnAngle)
```

Abstraction is good!

Problem solving: Draw a circle with a given radius

- *Notice: a polygon with many sides looks like a circle*
  - But how many sides to draw?
  - And how long should each side be?
- Start simple: decide to draw 360 sides every time
- Think: length of 1 side = circumference / 360
  - And remember from math that circumference equals $2\pi r$
- Put it all together: Listing 1.7 from the text (p. 40)

```
def drawCircle(myTurtle, radius):
    circumference = 2 * 3.1415 * radius
    sideLength = circumference / 360
    drawPolygon(myTurtle, sideLength, 360)
```

Try it!

Next

Finding π (and some more basic techniques)