CCNYPSY31170

How do computers work and what is a computer program?

Greek philosopher Aristotle explored the question of how we could prove that a statement was true. He identified a few basic concepts including

"if something is true it can NOT also be false (and vice versa)", and

"if it is true that when A and B are both true then C is true, then if A is true AND B is true, then C must be true", and

"if C is true when either A OR B is true, and A is true, then C is true, and also if B is true then C is true, or if both A and B are true, then C is true also."

There are other similar relationships, but you get the idea, I hope. This use of TRUE, FALSE, AND, OR, XOR (A or B is true, but not both) and NOT is the foundation of logic, and from it all mathematics can be derived.

Because "true" and "false" are just words representing concepts, we can also use the symbols 0 (zero) and 1 (one) to represent false and true, allowing us to use base 2 (binary) arithmetic to describe logical and mathematical relationships. For example in binary, 1 + 0 equals 1, and 1 plus 1 equals 10 because in binary notation, the last (rightmost) place is the one's place (how many 1's; either 0 or 1), and the next place to the left is the 2's place (how many 2's; either 0 or 1) and so on, which each position from right to left being twice as many 2's as the previous. For example, the binary number 10101 means the same as the decimal number 21, because it indicates one 1 (rightmost place), plus no 2's, plus one 4, plus no 8's plus one 16, which totals to 16+4+1 or 21 in decimal notation. Binary numbers can also serve as codes representing text characters, enabling words, letters and symbols to represent human language content.

It's annoying to humans to use binary arithmetic, but as engineers developed electrical circuits in the years before World War II, they realized that circuits could be set up like binary numbers, where the presence of electricity meant 1 (or true) and the absence of it meant 0 (or false). By using one wire to turn an electrical switch from off to on, or vice versa (i.e. to "write" a one or zero in the circuit), and another wire to test if there was electricity coming out of the switch (i.e. to "read" if there was a one or zero), we could represent logic (and arithmetic, and text) in electrical circuits. Cool. But it wasn't until after WWII that electrical engineering got advanced enough to build complex enough circuits to make useful logical and arithmetic calculations using just electrical switches.

The emergence of "computers" in the post-war period brought the theory and electrical engineering advances together, creating "hardware": machines made of circuits and other components, and "software": arrangements of binary digits that defined and controlled what the hardware did.

1. **Computer hardware**
    1. parts of a computer: CPU, RAM, persistent storage, peripherals;
    2. input and output: keyboard, mouse, display, printer, data transfer via wired network and wireless (radio) transmission;
    3. processing: arithmetic and logical calculations done in electrical circuits, moving results into and out of RAM, using binary numbers representing integers, floating decimal point numbers and as character symbols.

2. **Software**
    1. operating system (e.g. Windows, Linux, Android, etc.): tells the hardware what to do;
    2. programming language (e.g. Python, C++, Java, etc): a program that tells the operating system and CPU what to do;
    3. data: stored bits (binary digits) that represent information, including the programs' instructions;
    4. algorithms: abstract descriptions defining what to do with the data;
    5. programs are a special kind of data processed by the programming language (itself a program), implementing the algorithms as specific operations spelled out in the programming language syntax that the language can execute by converting the program into binary digits (on and off switches) in the electrical circuitry.
    6. The basics of all computer programming are specifications of sequential (one step after another), conditional (if something is true do this, otherwise do something else), and iterative (repeat a sequence until a condition is true) operations acting on data.

The central processing unit (CPU):

registers: circuits that hold binary sequences currently being used

i**nstruction pointer**: holds the location in memory of the next instruction

the arithmetic and logical unit (ALU): does the next instruction using the binary numbers in the registers and in RAM, and places the result in the accumulator

accumulator: holds interim results of instructions

The random access memory (RAM):

binary circuits that can be read from and written to (turned off and on) by the ALU as it executes instructions by changing the on-off states of electronic switches in the circuitry.

Machine language is the binary number codes that cause the ALU circuits to perform the actual logical and arithmetic binary number calculations. These include the AND, OR, XOR and NOT logical operations as defined by Aristotle back in the day. They also include MOV, moving data to and from the RAM and the CPU registers. They also include reading and writing to the instruction pointer to control where the next instruction comes from. Normally, by default the instruction pointer is advanced by one (to the next instruction), but it may also be set to some other location to jump to another part of the program. **Assembly language** is the transcription of binary machine language into human readable text acronyms, making it much easier to read and write machine-level programs by hand. Each type or brand of computer CPU has its own instruction set that depends on its specific hardware circuits, so machine language that runs on one type of CPU will not run on another.

Programming languages are programs that provide sets of instructions that allow programmers to work at a higher, more abstract level to define the algorithms. The statements written in the programming language are converted by the language program into machine language, either all at once (called "compiling") or one statement at a time (called "interpreting"). Some languages first pre-process the programs into a compressed version that is more efficiently translated into binary machine codes. This is called pseudo-compiling, and Python and Java are two examples of this. One advantage of higher level programming languages is that they can be made to convert their instructions into various different machine languages, enabling one high level language program to run on many types of computer.

Data structures: numbers, words, symbols, etc. that are related or associated with one another usually need to be organized so as to make their relationships obvious and to facilitate finding and storing the data items that collectively mean something, and therefore constitute information. These forms of organization are implemented in computer languages in several common forms, including:

databases: data organized into individual records that all have the same combination of attributes, such as an employee database holding a record for each employee with a name, address, SS#, etc. Often an attribute of a record may be another set of record in the database, such as the employee record including a database of payroll information including records of time worked and

amount paid, etc. The most common form of database used today is created, accessed and written using SQL (Structured Query Language).

spreadsheets: data, usually numeric, organized into rows and columns, such as Excel.

text files: collections of text (called files) each given a name, stored individually as a collection of binary coded text data, usually in hierarchically organized groups of files called directories.

binary files: collections of raw binary data such as image files and audio files, each type of which has its own file format defining how to interpret the raw data.

In addition to these varieties of data structures that are stored externally on permanent or non-volatile memory devices such as disk drives, programming languages provide a variety of temporary data structures to hold information as it is processed.

Python data structures

Python has several types of data that programmers can use to store and reference and manipulate in the course of a program: **numbers**, of course, including integers, floating point numbers and complex numbers, and text, including **characters** in the major coding systems such as ASCII and Unicode enabling text in various languages other than English that have non-English punctuation marks and other characters, as well as entire alphabets for other languages. Individual characters of text can be put together and surrounded by quote marks to create the data type, "**text string**" i.e. a string of characters. Text strings are essentially lists of characters.

Python considers the **list** as a basic data type also, represented by numbers, characters and text strings surrounded by square brackets and separated by commas, e.g. ["this", "is", "a", "list of" ,7, "items"] . Lists can have other lists included in them as items, e.g. ["a", ["list", "of", 3], "items embedded in a larger list also having 3 items"]. Python provides the programmer with built in abilities to access specific items in a list, by the items' positions in the list, using square brackets to indicate the desired positions . Python uses zero-based indexing so [0] is the position of the first item, e.g. ['Dave', 'Britton'][0][0] indicates the character "D" since D is the starting (zero-th) item of the zero-th item in the overall list. "Dave" is the first item in the main list, and "D" is the first item in the text string list of characters "Dave". This is called "slicing" and makes it easy to program text manipulations in Python.

Another important data structure in Python is the "dictionary". Dictionaries allow data to be retrieved by a name or label instead of by position. A Python dictionary is enclosed in curly braces { } within which there is a list of labels (called "keys") and matching data values that the keys reference, e.g. {'1':'A', 'B':2, "c":'3'}. If we assign a variable to have this dictionary as its value, ie, x={'1':'A', 'B':2, "c":'3'} we can obtain the data values by key, rather than position: x['1'] will evaluate to 'A', and x["B"] will evaluate to 2. Being able to find data by name instead of by location or position in a list can be very useful, and Python conveniently allows the keys and values to mix text and numbers.

Python also provides data structures called “objects” that combine data and algorithms. These are created by the programmer defining a “Class” that has both programmatic computation functions and named attributes that hold data. For example, one could define a Class called “Person” that has data attributes named “firstname”, “lastname”, “paymentmethod”, and “amount”. The Person class could also have a function named “paybill” that generates a payment of “amount” in the person’s name, using the “paymentmethod” of “cash”, “check” or “creditcard”. To create a Person object, a variable is assigned to be of the class Person, and its data attributes are specified. When it comes time to pay a bill, that instance of Person determines how to make the payment depending on its “paymentmethod” attribute.

If we define the class to have the function paybill and the other attributes specified above, the program code looks like this:

```
Class Person
  __init__( self, fname, lname, paytype=”cash”, amount=0)
        self.firstname = fname
        self.lastname = lname
        self.paymentmethod = paytype
        self.amount = amount

        def paybill(self, amount):
                self.amount = amount
                print ( self.firstname, “paid”, self.amount, “with”, self.paymentmethod)
```

Now we can create the object Joe:

```
Joe = Person(“Joseph”, “Brown”, “cash”,0)
```

This creates an object of the class Person and assigns it to variable Joe (which is technically called instantiating an instance of the class Person).
When it’s time for Joe to pay a bill of billamount =”$5”, we tell the object to run the paybill function using this line of code:

Joe.paybill(billamount)

and the paybill function returns:
“Joseph paid $5 with cash”

We don’t need to know Joe’s preferred payment method at the time the payment is made, because it was already set when Joe was created. If Joe later changes his mind and wants to pay by check, we execute one line:
Joe.paymentmethod = “a check”

Then his next payment displays:
“Joseph paid $5 with a check”
We don’t have to re-program the payment function, it is built into the Person class, so all Person objects can pay in whatever method is chosen. Even better, we can define new classes of Persons that inherit all the functions and attributes of the base class, but can have new attributes and functions added just for them, or an inherited function or attribute changed. This ability of “objects” to inherit properties from other objects saves lots of repetitive coding for just small variations in functionality.

This a simplistic example, but it illustrates the point that “object-oriented programming”, as this is called, can make programs more efficient and more flexible by combining algorithms with related data structures in a single programming construct, the Class, that can be used in larger programs with little or no additional programming, and which keeps together data and functions that are connected, facilitating collaboration and software updating, especially in large systems.

The Social Context of Computing

After WWII, computers began to be commercially available, although they were very large and expensive, limiting them to big companies and academic institutions with adequate funding. IBM rapidly came to dominate the market, due to their experience selling business equipment to large companies. Because IBM had a virtual industry monopoly, the development of commercial software was largely limited to their product development, and they bundled their computers and software together, leaving no room for competition in the commercial software market. However, the academic computing environment grew rapidly as interest in, and accessibility to, computers permitted experimentation and development of new concepts, new techniques and new computer languages.

Academic programs often supported IBM's much smaller competitors, such as Control Data Corporation, Amdhal, Burroughs and Honeywell, (and IBM even sometimes supported the smaller competitors to keep them alive so IBM itself could avoid anti-trust laws and regulations). Cold-war era U.S. government military agencies (such as the Naval Defense Advanced Research Projects Administration, aka. DARPA) provided significant grants to universities to conduct research and development of computer technology, believing this would support basic research needed to foster the development of military applications of computers. For example, a project to connect the academic computers across the country, to protect critical communications in the event of a nuclear war causing the loss of stand-alone individual systems, led to the development of computer networking software and hardware that became known as DARPA-net (since DARPA funded it), but eventually in the 1990's expanded to become the Internet we use today.

Ironically, despite IBM's closed proprietary software product line, the single most important software development for the business world during this time was the design and development of the COBOL computer language that came to dominate the business software environment; but COBOL came not from IBM but from the work of Naval Reserve officer Grace Hopper at Harvard University. Hopper conceived of and developed the idea of having an English-like computer language program that could translate English commands (defining data processing functions) into machine language – that is, the first compiler. COBOL was free and open source, coming from a university academic background that was in part government funded, but it was so good for straightforward business data processing that commercial versions of the language were eventually developed to run on various different makes of computers (especially compared to the difficult assembly language programming often favored by IBM) and it became the language of choice for business computing, at least until the advent of "minicomputers" in the late 1970's, and even into the 1990's.

Digital Equipment Corporation (DEC) developed a new breed of computers during the 1960's that were much smaller and less expensive than those of IBM, using improvements in electronic circuit design and manufacturing. By the late 1960's the minicomputers were opening up new markets selling to smaller businesses and also to academic institutions. DEC computers had 16-bit CPU processors and were usually programmed in the simple computer language BASIC that was especially designed for individual interactive use at a 24 line x 80 column video terminal. Bigger 32-bit CPU IBM "mainframe" computer systems still used punch cards and batch processing, so the nimbler DEC user

experience was attractive to many researchers and more affordable for businesses. Most business applications had to be custom-programmed, so the industry of computer programming (now called “software development”) grew rapidly to meet the increased demand.

In 1975 the IMSAI and Altair personal computer kits came on the market. These used the inexpensive Intel 8080 8-bit processor and had open plug-in electronic slots to allow add-on peripherals. They were intended for the do-it-yourself hobbyist market, since the buyer had to solder the components into a printed circuit board called the “motherboard”. They opened a radical new possibility for individuals to own their own fully functional computer. Soon electronics hobbyist companies such as Tandy (Radio Shack) and consumer electronics manufacturers such as Zenith introduced fully assembled personal computers, which became called “microcomputers” since they were smaller than the “minicomputers” New companies formed during the late 1970’s and early 1980’s to design, produce and market these small, inexpensive 8-bit “personal computers”, including Apple, whose Apple-II became very popular with accountants who could use it to electronically create and interactively modify spreadsheets, which had been an important but very tedious manual task before this. Word processor programs such as Electric Pencil and WordStar enabled these low-cost personal computers to have the functionality of very expensive dedicated word processor systems such as those made by Wang, and the personal computer revolution was underway.

In 1983 a small R&D group within IBM announced the upcoming availability of a microcomputer they called the “IBM PC” that would have a larger 16-bit processor, the Intel 8086. Everyone assumed the IBM PC would dominate the microcomputer market since IBM dominated the mainframe world, but soon competitors, also offerring new 16-bit PC systems that were compatible with the IBM PC software and add-on peripherals, entered the market, while specialized computer stores opened across the country to make PC’s readily available to individuals and small businesses.

NEXT: The “Free and Open Source” Movement begins (“free as in liberty, and free as in beer...”)