UNIVERSITÀ DEGLI STUDI FIRENZE

Ki.Mo.Co Kinesiology and Motor Control Lab

# Programming

General Background

## Machine Code

A computer is **NOT** a person.

If we wish to communicate something to a computer, we need to do it in a way that the computer will understand.

**Hello**

my name is

01010100 01111001 01110010
01100001 01111001

## Machine Code

Binary code, consisting of 1 and 0, comes from electrical engineering. This is basically a way to say whether electrical current is flowing (1) or not (0).

In other words binary code refers to the state as either on (1) or off (0)

# Machine Code

In the Decimal system, each position of a digit represents a power of 10.

| **1** | **2** | **4** | **3** | **9** |
|---|---|---|---|---|
| $10^4$ | $10^3$ | $10^2$ | $10^1$ | $10^0$ |
| 10,000 | 1,000 | 100 | 10 | 1 |

When we read a number we therefore look at the number of 1's (in this case 9), the number of 10's (e.g., 3) etc.

## Machine Code

In a binary system each digit position represents a power of 2 (4, 8, 16, etc.).

| **1** | **1** | **0** | **1** | **0** |
|---|---|---|---|---|
| $2^4$ | $2^3$ | $2^2$ | $2^1$ | $2^0$ |
| 16 | 8 | 4 | 2 | 1 |

When we read this number we can see that we have one 16, one 8 and one 2, which would be 16+8+2 = 26

## Machine Code

We use powers of 2 because of the way the components in the computer (mainly transistors) operate. That is, they can have only 2 states, i.e., on or off.

## Binary Code

Each digit of a **binary code**, is called a binary digit, or **bit**.

Since each bit can only convey information about a single state, on or off, it is not too useful.

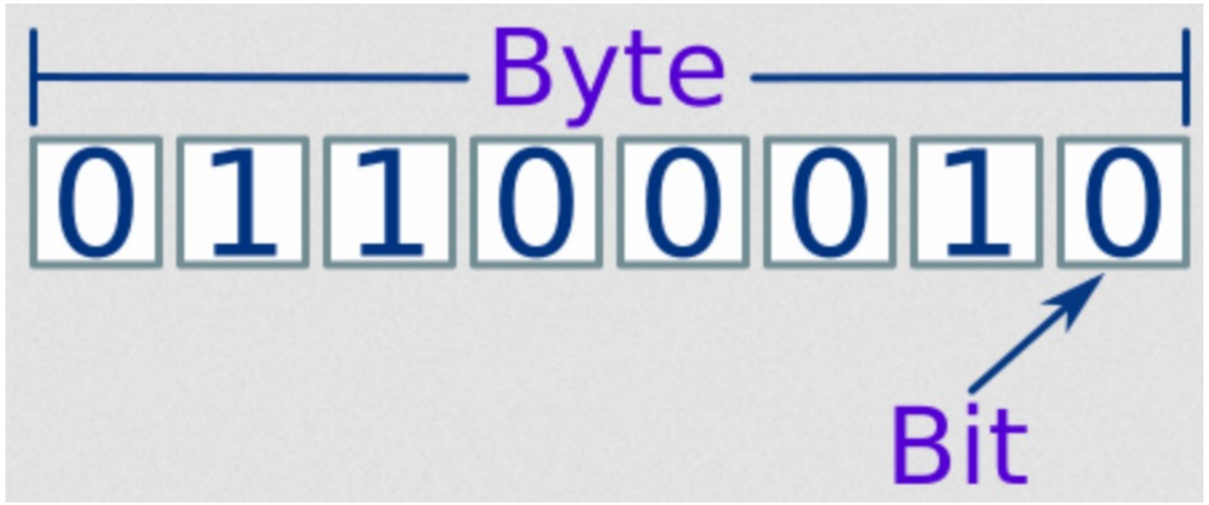
However, when we combine them into groups, we can convey much more information.

## Binary Code

When 8 bits are grouped together, we call this a **byte**.

The amount of possibilities of binary code in one byte is the equivalent to $2^8$, that is 256.

This is enough to cover all of the alphabet, as well as special characters.


Byte
0 1 1 0 0 0 1 0
Bit

| Abbr. | Prefix name | Decimal size | Size in thousands | Binary approximation | Address variable size |
|---|---|---|---|---|---|
| K | kilo- | $10^3$ | 1,000 | $1{,}024 = 2^{10}$ | 10 |
| M | mega- | $10^6$ | $1{,}000^2$ | $1{,}024^2 = 2^{20}$ | 20 |
| G | giga- | $10^9$ | $1{,}000^3$ | $1{,}024^3 = 2^{30}$ | 30 |
| T | tera- | $10^{12}$ | $1{,}000^4$ | $1{,}024^4 = 2^{40}$ | 40 |
| P | peta- | $10^{15}$ | $1{,}000^5$ | $1{,}024^5 = 2^{50}$ | 50 |
| E | exa- | $10^{18}$ | $1{,}000^6$ | $1{,}024^6 = 2^{60}$ | 60 |

## Binary Code

A **binary code signal** is a series of electrical pulses that represent numbers, characters, and operations to be performed.

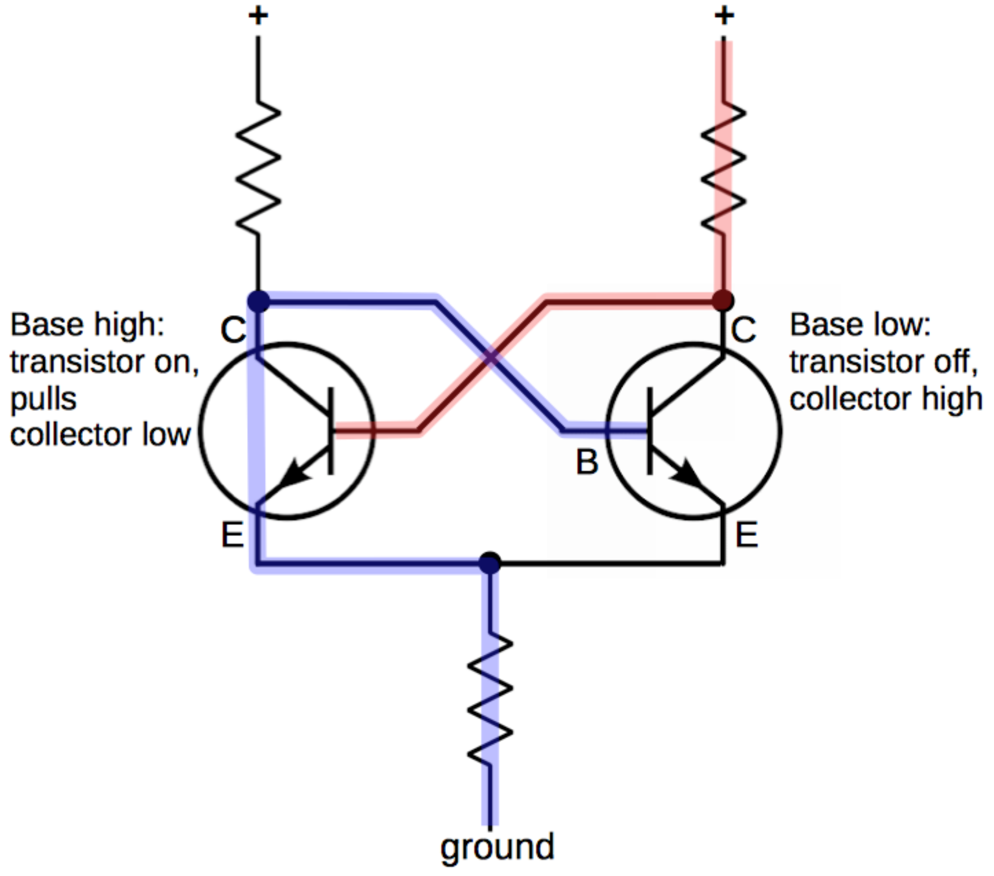
A device called a clock sends out regular pulses, and components such as transistors switch on (1) or off (0) to pass or block the pulses.

## Binary Code

We can use binary code to both store information as well as to execute commands.

The ability to store information was one of the biggest turning points in terms of computer development. This was made possible by the inventions of transistors and logic gates.

Using these components, it is possible for a machine to “remember” its previous state



Example of a basic latch mechanism with two transistors

## Basic Operations in Binary Code

Apart from storing information, we can also make some basic operations with binary code.

Mainly addition, subtraction, multiplication and division.

|  | Addition | Subtraction | Multiplication | Division |
|---|---|---|---|---|
| i) | 0 + 0 = 0 | 0 – 0 = 0 | 0 x 0 = 0 | 0 /1 = 0 |
| ii) | 0 + 1 = 1 | 1 – 0 = 1 | 0 x 1 = 0 | 1 / 1 = 1 |
| iii) | 1 + 0 = 1 | 1 – 1 = 0 | 1 x 0 = 0 | 0 / 0 = not allowed (not valid) |
| iv) | 1 + 1 =10 | 1-0 = 10 – 1 (with borrow 1) = 1 | 1 x 1 = 1 | 1 / 0 = not allowed (not valid) |

## Representing Negative Numbers

Obviously, representation of only positive integers is not sufficient for mathematical applications.

To overcome this issue, the **two’s complement** system was invented.

Basically, to get the a negative number in binary, we take the number we are interested in, invert the digits and add one.

The result is the negative complement of the number.

## Representing Negative Numbers

Let’s use number 28 in 1 byte as an example, which in binary:

00011100

We invert the digits (1→0 and 0→1):

11100011

We add one and we get (-28):

**11100100**

## Representing Negative Numbers

Why does this work?

It is basically simple arithmetic applied to binary. Meaning, if we were to subtract 19 from 21

$$\begin{array}{r} 21 \\ -\ 19 \end{array}$$

We treat 1 – 9 as 11 – 9, we carry a 1 from the next position.

## Representing Negative Numbers

Why does this work?

```
  1
  21
- 19
   2
```

We the subtract 1 from the next position and continue with the subtraction as normal.

## Representing Negative Numbers

Why does this work?

$$\begin{array}{r} 0 \\ -\ 19 \\ -\ 19 \end{array}$$

Using the same concept, when we represent a negative number, it is the same as subtracting from 0.

# Representing Negative Numbers

Why does this work?

If we are stupid (like a computer), subtracting from 0 will look like this:

```
                  1            11           111          1111
000000        000000        000000        000000        000000
-    3        -    3        -    3        -    3        -    3
------        ------        ------        ------        ------
                   7            97           997          9997
```

# Representing Negative Numbers

Why does this work?

We can do the same thing in binary (0 - 75):

```
                    1            11           111          1111
  00000000      00000000      00000000      00000000      00000000
- 01001011    - 01001011    - 01001011    - 01001011    - 01001011
----------    ----------    ----------    ----------    ----------
                       1            01           101          0101
```

```
    11111        111111       1111111      11111111
  00000000      00000000      00000000      00000000
- 01001011    - 01001011    - 01001011    - 01001011
----------    ----------    ----------    ----------
     10101        110101       0110101      10110101
```

## Representing Negative Numbers

Why does this work?

If we work with 8 bits, subtracting from 0, will be the same as subtracting from a 9 bits with one at the beginning.

```
  11111111
  100000000
- 01001011
----------
  010110101
```

## Representing Negative Numbers

Why does this work?

This would also be the same as just working with 8bits and adding 1

```
 11111111
 100000000
- 01001011
----------
 010110101
```

```
          1
+ 11111111
- 01001011
----------
  10110101
```

## Representing Fractions

Aside from working with integers (negative or positive) often we would like to use fractions of numbers.

Using binary code, we simply need to adapt the way we interpret the code.

# Representing Fractions

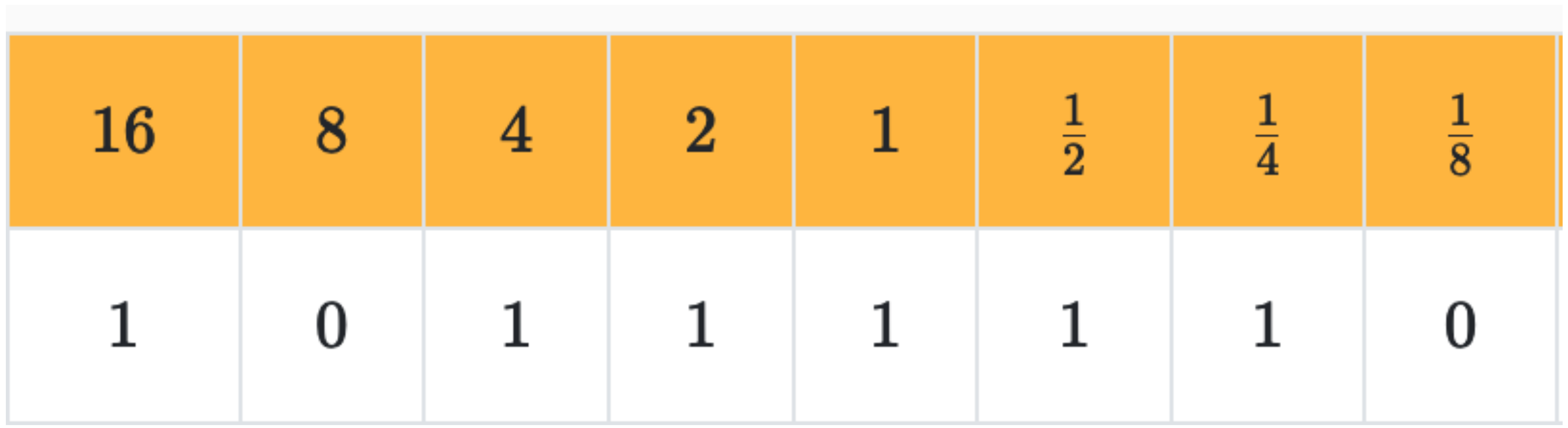
We can just “shift” our positional values of any binary signal to values smaller than $2^0$:

| $2^3$ | $2^2$ | $2^1$ | $2^0$ | $2^{-1}$ | $2^{-2}$ | $2^{-3}$ |
|---|---|---|---|---|---|---|
| **8** | **4** | **2** | **1** | $\frac{1}{2}$ | $\frac{1}{4}$ | $\frac{1}{8}$ |
| 0 | 1 | 0 | 0 | 0 | 1 | 0 |

# Representing Fractions

Following this, we just proceed by adding the numbers as a whole, just as before.

In this case it would be ¼ + ½ + 1 + 2 + 4 + 16 = 23 ¾ = 23.75



| 16 | 8 | 4 | 2 | 1 | $\frac{1}{2}$ | $\frac{1}{4}$ | $\frac{1}{8}$ |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 |

## Representing Fractions

This way of working is called **fixed-point** representation.

Obviously it presents some limitations, as the name suggests, it is fixed, meaning that we have to use the same ”shifting” when working with the code.

Also, there is a bit of a compromise between the amount of decimal to integers position (considering for example 8bits).

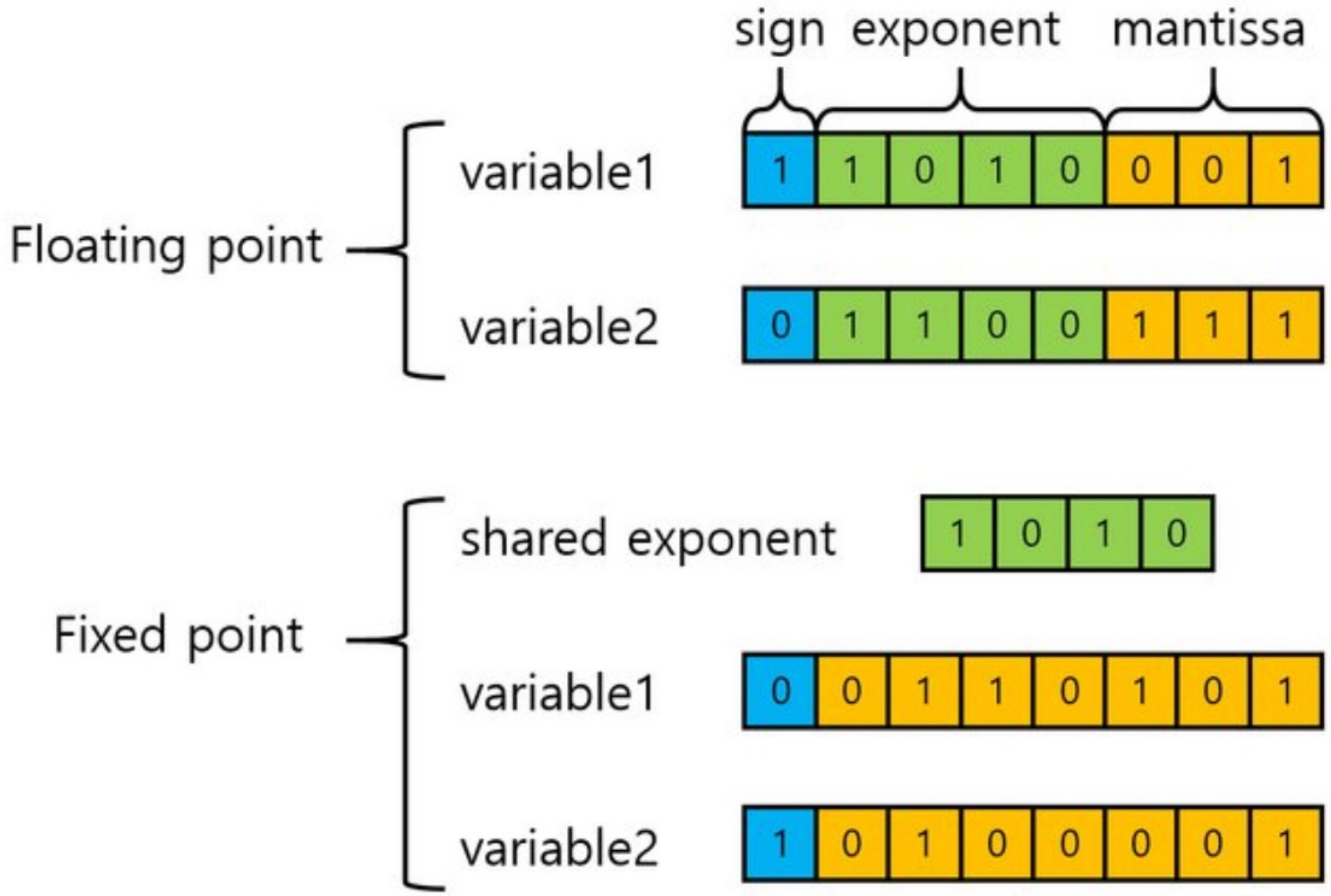
This can cause quite a lot of problems.

## Representing Fractions

To overcome this issue, and also the issue of very large and very small numbers, the concept of **floating-point** was introduced.

This allows the way that the **bits are allocated** to vary so that both very large and very small numbers can be represented.

## Representing Fractions



## Representing Fractions

The floating point approach is similar to **scientific notation** or **standard form** representation.

For example, the number 0.0007910 can be written in standard form as $7.9 \times 10^{-4}$.

This would be converted to 7.9 as the mantissa and $10^{-4}$ as the exponent.

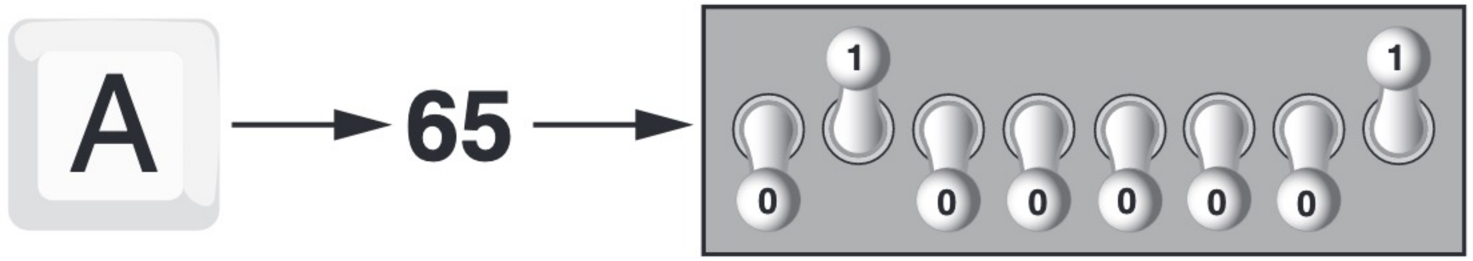
## Representing Characters

So far we have looked at numbers, suppose we would like to work also with characters. Since everything is eventually converted to binary code, we need to find a system that is able to represent characters in binary code.

The “American Standard Code for Information Interchange” (or **ASCII**) does exactly this.

## ASCII

ASCII is a set of 128 numeric codes that represent the English letters, various punctuation marks, and other characters.



| Decimal | Binary | Octal | Hex | ASCII | Decimal | Binary | Octal | Hex | ASCII |
|---|---|---|---|---|---|---|---|---|---|
| 64 | 01000000 | 100 | 40 | @ | 96 | 01100000 | 140 | 60 | ` |
| 65 | 01000001 | 101 | 41 | A | 97 | 01100001 | 141 | 61 | a |
| 66 | 01000010 | 102 | 42 | B | 98 | 01100010 | 142 | 62 | b |
| 67 | 01000011 | 103 | 43 | C | 99 | 01100011 | 143 | 63 | c |
| 68 | 01000100 | 104 | 44 | D | 100 | 01100100 | 144 | 64 | d |
| 69 | 01000101 | 105 | 45 | E | 101 | 01100101 | 145 | 65 | e |
| 70 | 01000110 | 106 | 46 | F | 102 | 01100110 | 146 | 66 | f |
| 71 | 01000111 | 107 | 47 | G | 103 | 01100111 | 147 | 67 | g |
| 72 | 01001000 | 110 | 48 | H | 104 | 01101000 | 150 | 68 | h |
| 73 | 01001001 | 111 | 49 | I | 105 | 01101001 | 151 | 69 | i |
| 74 | 01001010 | 112 | 4A | J | 106 | 01101010 | 152 | 6A | j |
| 75 | 01001011 | 113 | 4B | K | 107 | 01101011 | 153 | 6B | k |
| 76 | 01001100 | 114 | 4C | L | 108 | 01101100 | 154 | 6C | l |
| 77 | 01001101 | 115 | 4D | M | 109 | 01101101 | 155 | 6D | m |
| 78 | 01001110 | 116 | 4E | N | 110 | 01101110 | 156 | 6E | n |
| 79 | 01001111 | 117 | 4F | O | 111 | 01101111 | 157 | 6F | o |
| 80 | 01010000 | 120 | 50 | P | 112 | 01110000 | 160 | 70 | p |
| 81 | 01010001 | 121 | 51 | Q | 113 | 01110001 | 161 | 71 | q |
| 82 | 01010010 | 122 | 52 | R | 114 | 01110010 | 162 | 72 | r |

## ASCII

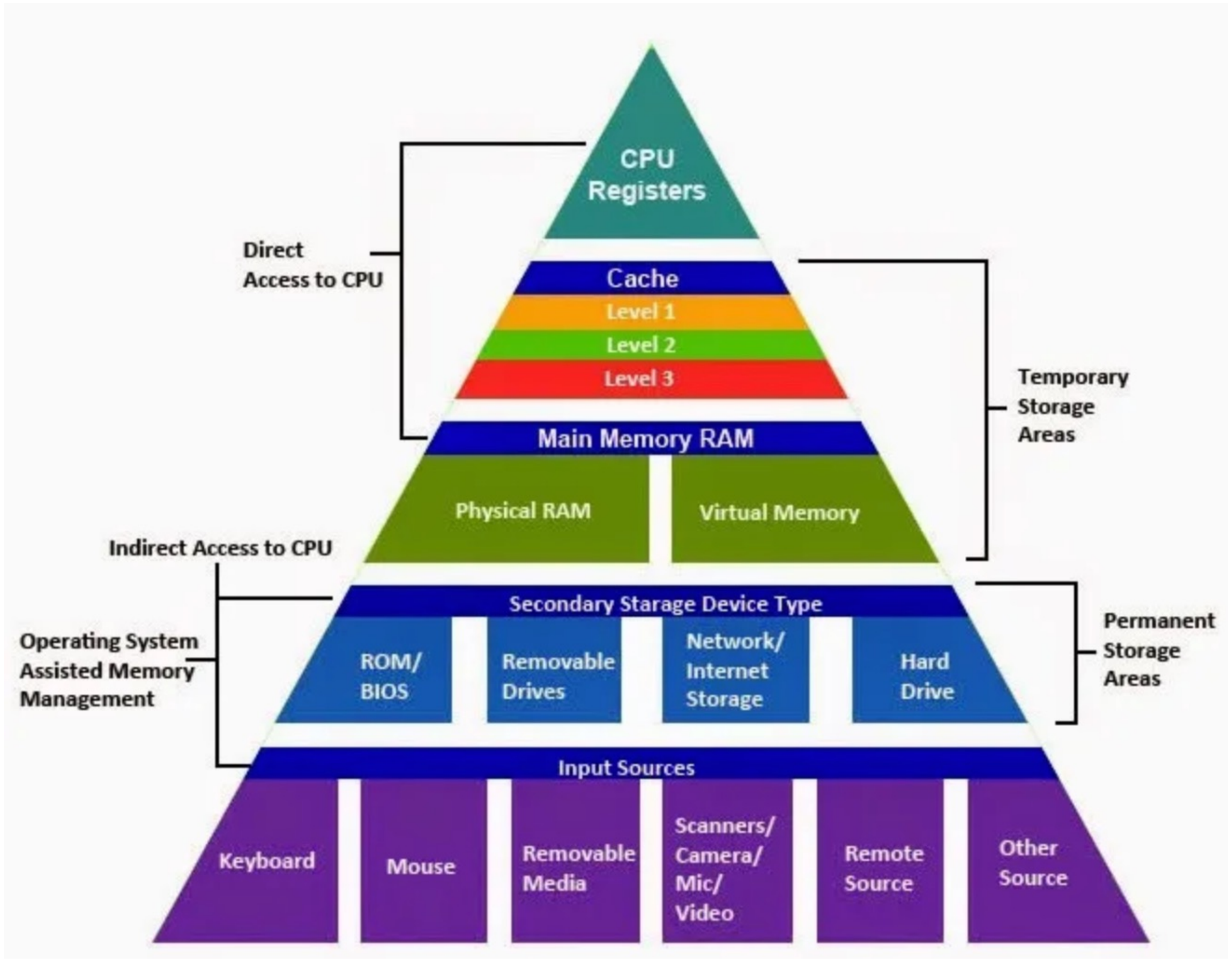
ASCII was widely used from the 1960’s to the 1990’s. Then it was slowly replaced by **Unicode**, which allows for more symbols and characters from different languages.

# How does it all work in a computer?

## Components

The main components of any computer consist of:

- CPU (Central Processing Unit)
- Main Memory
- Secondary Storage Devices
- Input Devices
- Output Devices


CPU Registers
Direct Access to CPU
Cache
Level 1
Level 2
Level 3
Temporary Storage Areas
Main Memory RAM
Physical RAM
Virtual Memory
Indirect Access to CPU
Secondary Starage Device Type
Operating System Assisted Memory Management
ROM/ BIOS
Removable Drives
Network/ Internet Storage
Hard Drive
Permanent Storage Areas
Input Sources
Keyboard
Mouse
Removable Media
Scanners/ Camera/ Mic/ Video
Remote Source
Other Source

## CPU

The central processing unit, or CPU, is the part of a computer that actually runs programs.

Each instruction in a program is a command that tells the CPU to perform a specific operation.

Here’s an example of an instruction that might appear in a program:

```
10110000
```

## CPU

Because the operations that a CPU knows how to perform are so basic in nature, a meaningful task can be accomplished only if the CPU performs many operations.

## CPU

When a CPU executes the instructions in a program, it is engaged in a process that is known as the **fetch-decode-execute** cycle.

1. **Fetch** read the next instruction from memory into the CPU.

2. **Decode** In this step the CPU decodes the instruction that was just fetched into machine code.

3. **Execute** The last step in the cycle is to execute, or perform, the operation.

## CPU

It is worth noting that everything from the CPU’s point of view is in binary.

1. **Fetch** – each memory unit has a unique binary address.

2. **Decode** – when information stored is compressed in some way (a shorter binary string), to save memory, we therefore need to decode it.

## CPU specs

What do the specifications of a CPU, e.g., 64-bit 3.9GHz, mean?

Processors work according to a clock that beats a set number of times per second, usually measured in gigahertz.

For instance, a 3.9-GHz processor has a clock that beats 3.9 billion times per second.

Each clock beat represents an opportunity for the processor to manipulate a number of bits equivalent to its capacity - 64-bit processors can work on 64 bits at a time.

## Main Memory

Main memory is commonly known as **random-access memory**, or **RAM**. It is called this because the CPU is able to quickly access data stored at any random location in RAM.

RAM is usually a **volatile** type of memory that is used only for temporary storage while a program is running. When the computer is turned off, the contents of RAM are erased.

## Main Memory

RAM is characterized by:

- **Bus Width** - the number of bits that can be sent to the CPU simultaneously

- **Bus Speed** - the number of times a group of bits can be sent each second

## Main Memory

These quantities define the efficiency of the memory, for example:

**100 MHz 64-bit** bus is theoretically capable of sending 8 bytes (64 bits) of data to the CPU 100 million times per second.

In reality, RAM doesn't usually operate at optimum speed. **Latency** changes the equation radically.

## Optimizing Memory and Performances

The problem that computer designers face is that memory that can keep up with a 1-gigahertz CPU is extremely **expensive** - much more expensive than anyone can afford in large quantities.

One way, among many, to alleviate bottlenecks in performances is by the use of **caching**.

## Cache

Caching is a technology based on the memory subsystem of your computer.

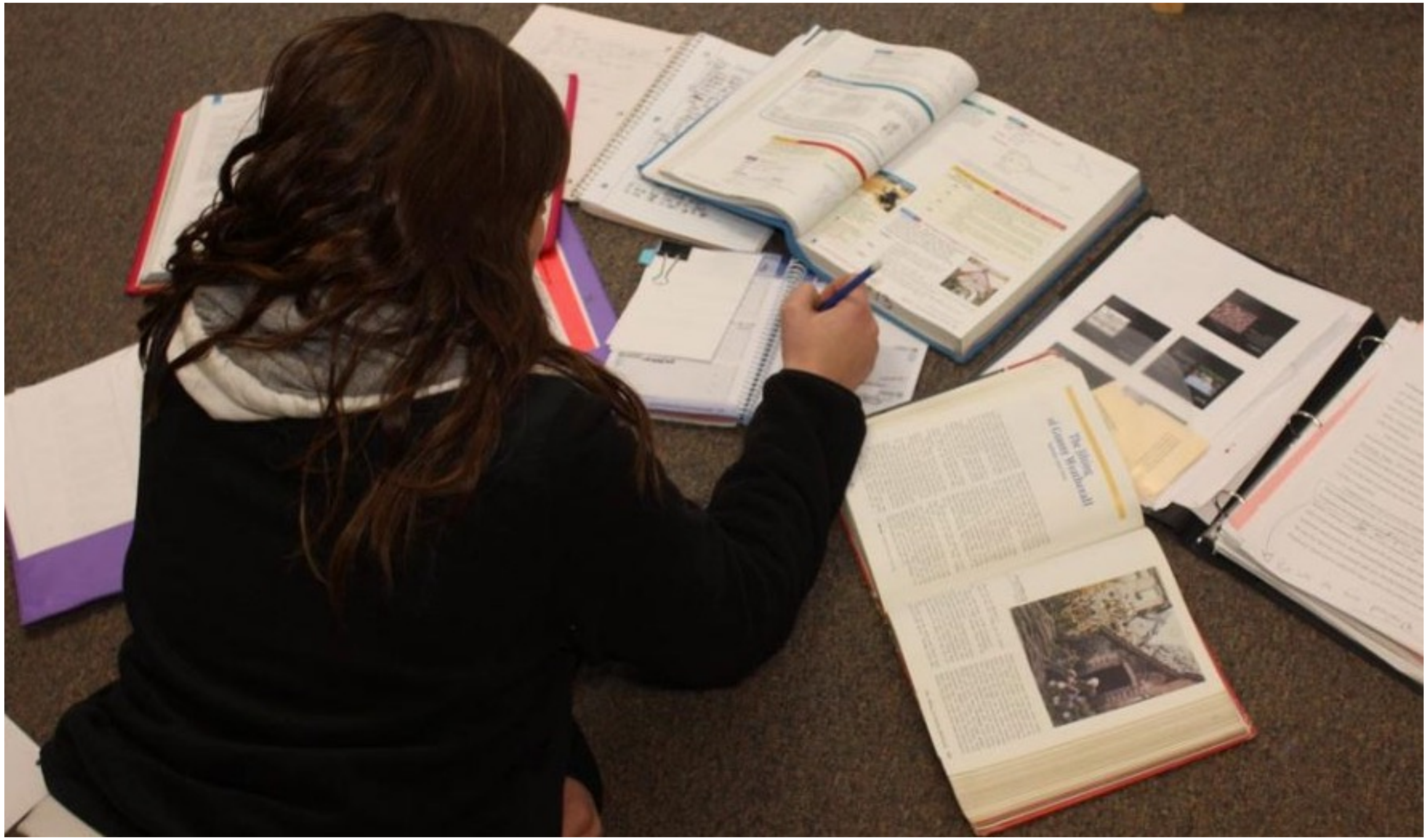
The main purpose of a cache is to accelerate your computer while keeping the price of the computer low.

Caching allows you to do your computer tasks more rapidly

## Cache

We use Caching pretty much all the time in our day to day lives.

The concept is very simple, if we use something, we want it to be readily accessible.



## Cache

When using a cache, you must check the cache to see if an item is in there. If it is there, it's called a **cache hit**. If not, it is called a **cache miss** and the computer must wait for a round trip from the larger, slower memory area.

There is a bit of educated guessing when it comes to caching..

## Putting it all together

## The Basic Scenario

You turn the computer on.

The computer loads data from **read-only memory** (ROM) and performs a power-on self-test (POST) to make sure all the major components are functioning properly.

The computer loads the **basic input/output system** (BIOS) from ROM. The BIOS provides the most basic information about storage devices, boot sequence, security, Plug and Play (auto device recognition) capability and a few other items.

The computer loads the operating system (OS) from the hard drive into the system's **RAM**.

## The Basic Scenario

When you open an application, it is loaded into **RAM**. To conserve RAM usage, many applications load only the essential parts of the program initially and then load other pieces as needed.

After an application is loaded, any files that are opened for use in that application are loaded into RAM.

When you save a file and close the application, the file is written to the specified storage device, and then it and the application are purged from RAM.

## Development of Computer Programming

## Assembly

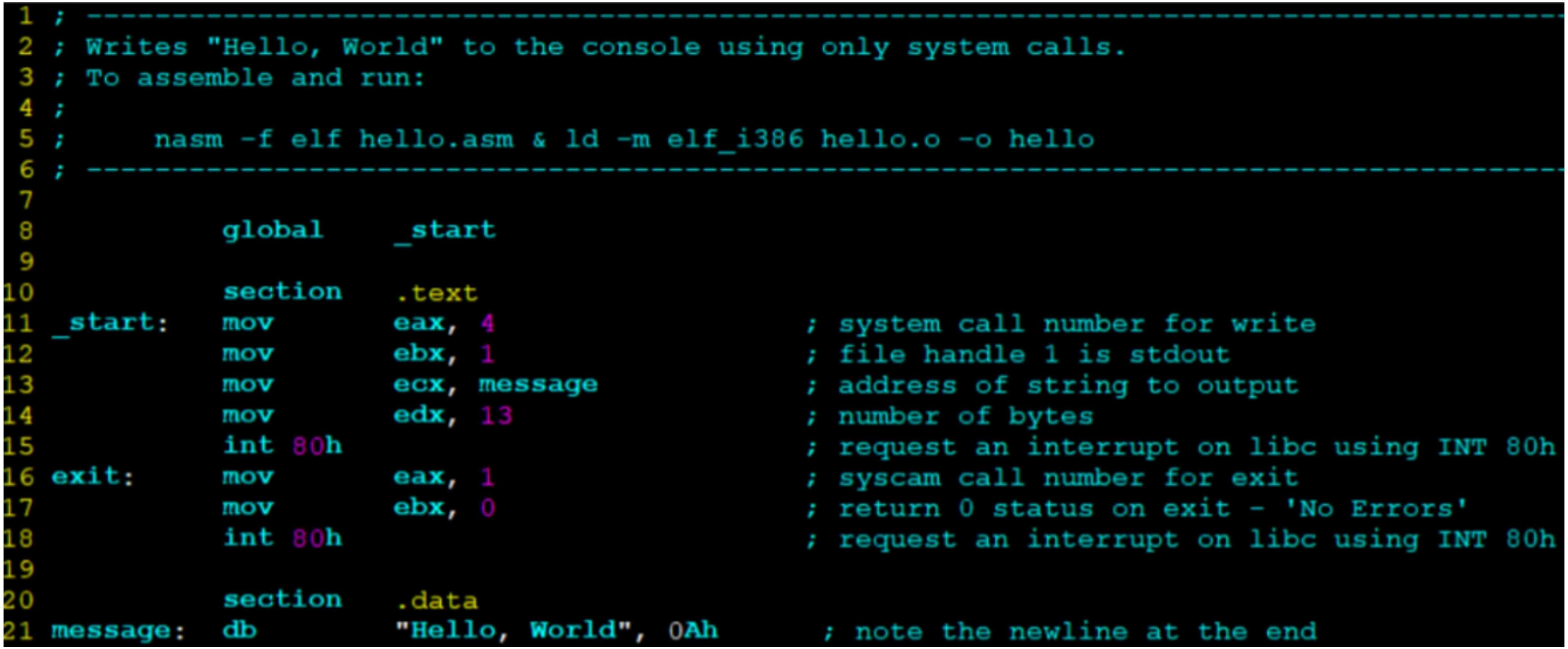
Writing binary instructions is very tedious and time consuming.

For this reason, **assembly** language was created in the as an alternative to machine language.

Instead of using binary numbers for instructions, assembly language uses short words that are known as **mnemonics**.

For example, in assembly language, the mnemonic **add** typically means to add numbers.

## Assembly

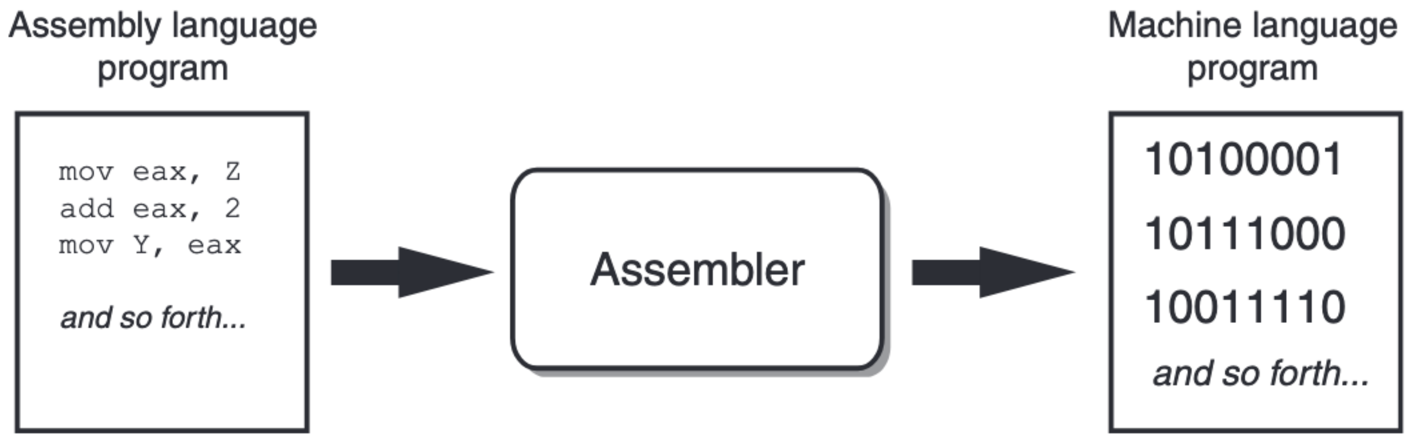


```
; ----------------------------------------------------------------------------
; Writes "Hello, World" to the console using only system calls.
; To assemble and run:
;
;     nasm -f elf hello.asm & ld -m elf_i386 hello.o -o hello
; ----------------------------------------------------------------------------

          global    _start

          section   .text
_start:   mov       eax, 4                  ; system call number for write
          mov       ebx, 1                  ; file handle 1 is stdout
          mov       ecx, message            ; address of string to output
          mov       edx, 13                 ; number of bytes
          int 80h                           ; request an interrupt on libc using INT 80h
exit:     mov       eax, 1                  ; syscam call number for exit
          mov       ebx, 0                  ; return 0 status on exit - 'No Errors'
          int 80h                           ; request an interrupt on libc using INT 80h

          section   .data
message:  db        "Hello, World", 0Ah      ; note the newline at the end
```

## Assembly

Assembly language programs cannot be executed by the CPU.

The CPU only understands machine language, so a special program known as an **assembler** is used to translate an assembly language program to a machine language program.



## Assembly

Assembly language is a direct substitute for machine language, and like machine language, it requires that you know a lot about the CPU.

Assembly language also requires that you write a large number of instructions for even the simplest program.

Because assembly language is so close in nature to machine language, it is referred to as a **low-level language**.

## High-Level Languages

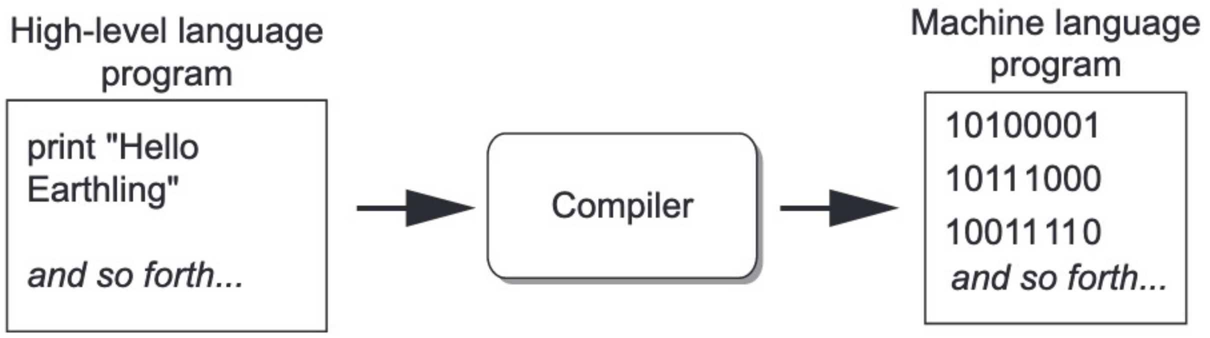
A high-level language allows you to create powerful and complex programs without knowing how the CPU works, and without writing large numbers of low-level instructions.

In addition, most high-level languages use words that are easy to understand.

## Compiler

A compiler is a program that translates a high-level language program into a separate machine language program.

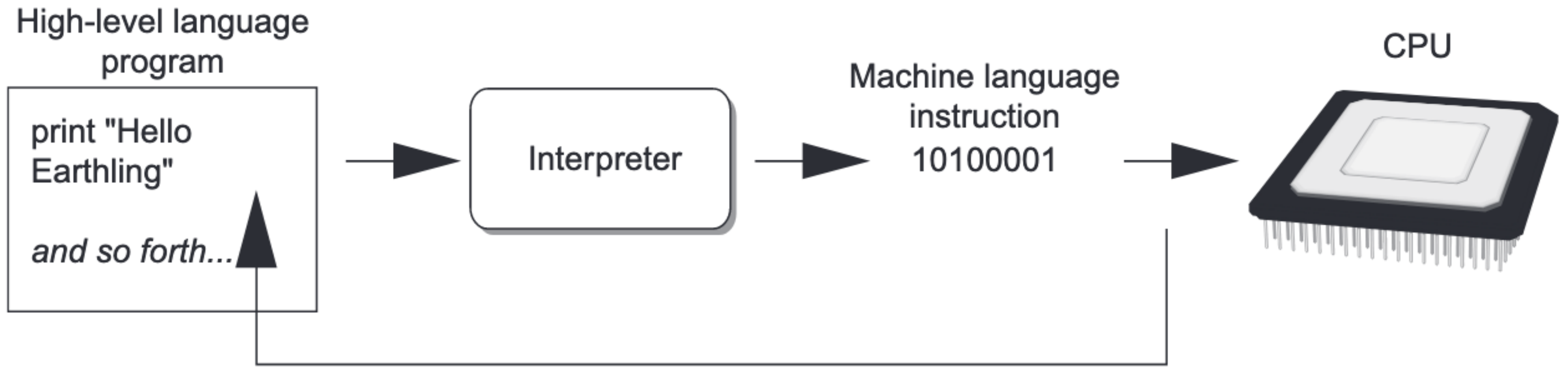
The machine language program can then be executed any time it is needed.

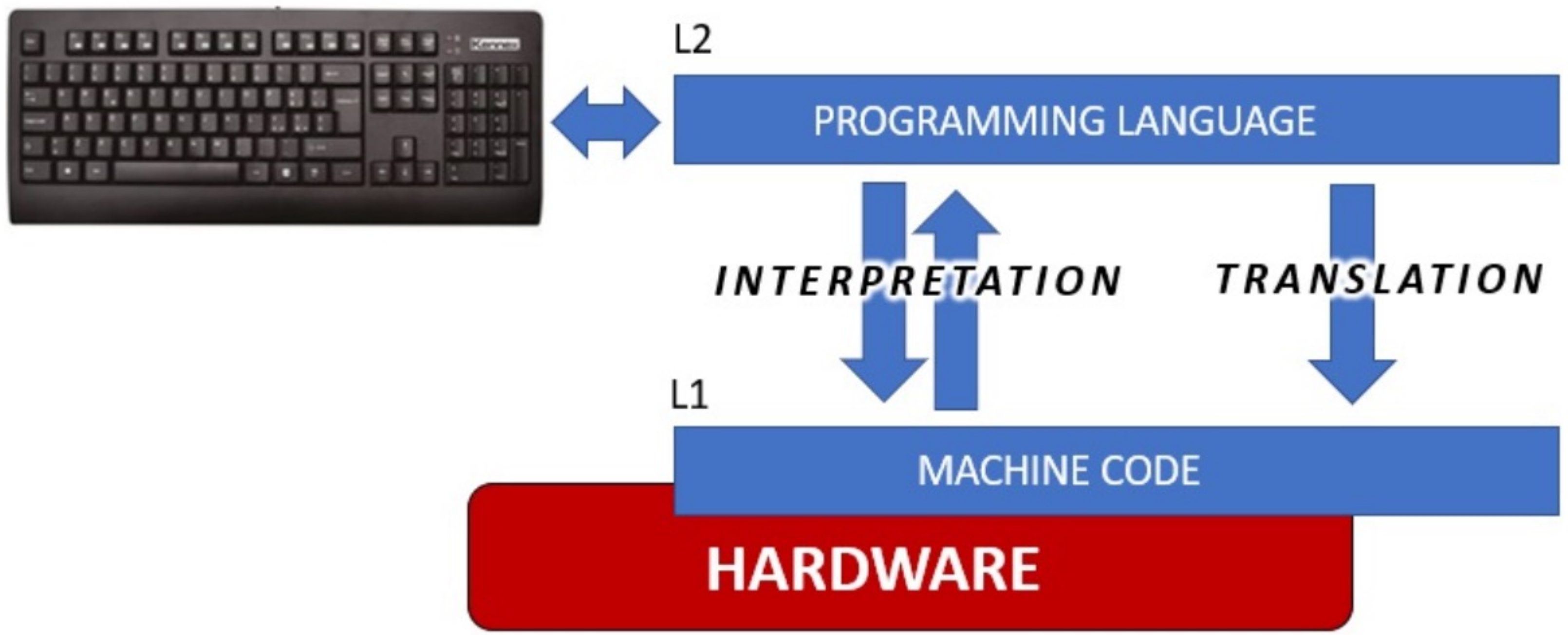


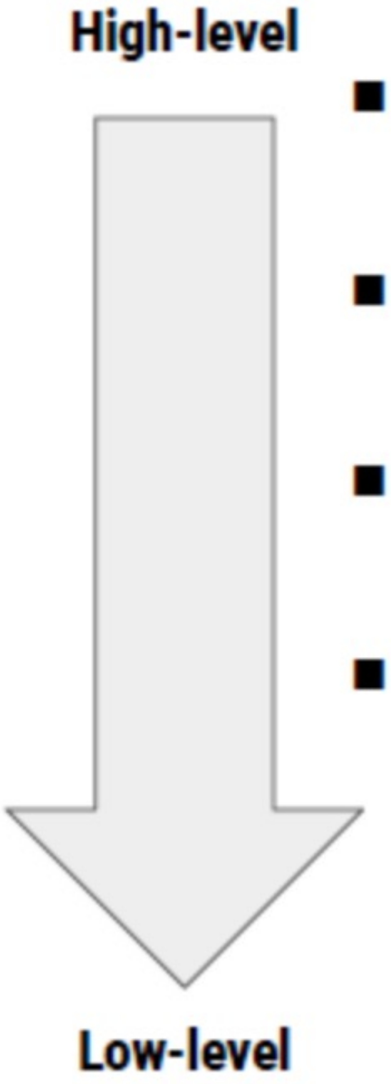
# Interpreter

Python uses an interpreter, which is a program that both translates and executes the instructions in a high-level language program.

As the interpreter reads each individual instruction in the program, it converts it to machine language instructions and then immediately executes them.




L2
PROGRAMMING LANGUAGE
INTERPRETATION
TRANSLATION
L1
MACHINE CODE
HARDWARE



- Python, JavaScript

  *Interpreted every time it runs*
- C, C++

  *Compiled into an executable file*
- Assembly language

  *Assembled into machine code*
- Machine code

  *Run by the CPU*

## C

C is considered to be a mid level programming languages.

For a long time, C was the dominating programming language and it is still used today.

By being a mid-level programming language, it offers some advantages compared to the higher level languages. Mainly, the code that is written is more stable and runs faster.

## C

When programming in C, we have to pay careful attention to the memory usage.

Meaning we need to allocate and free memory manually in the code.

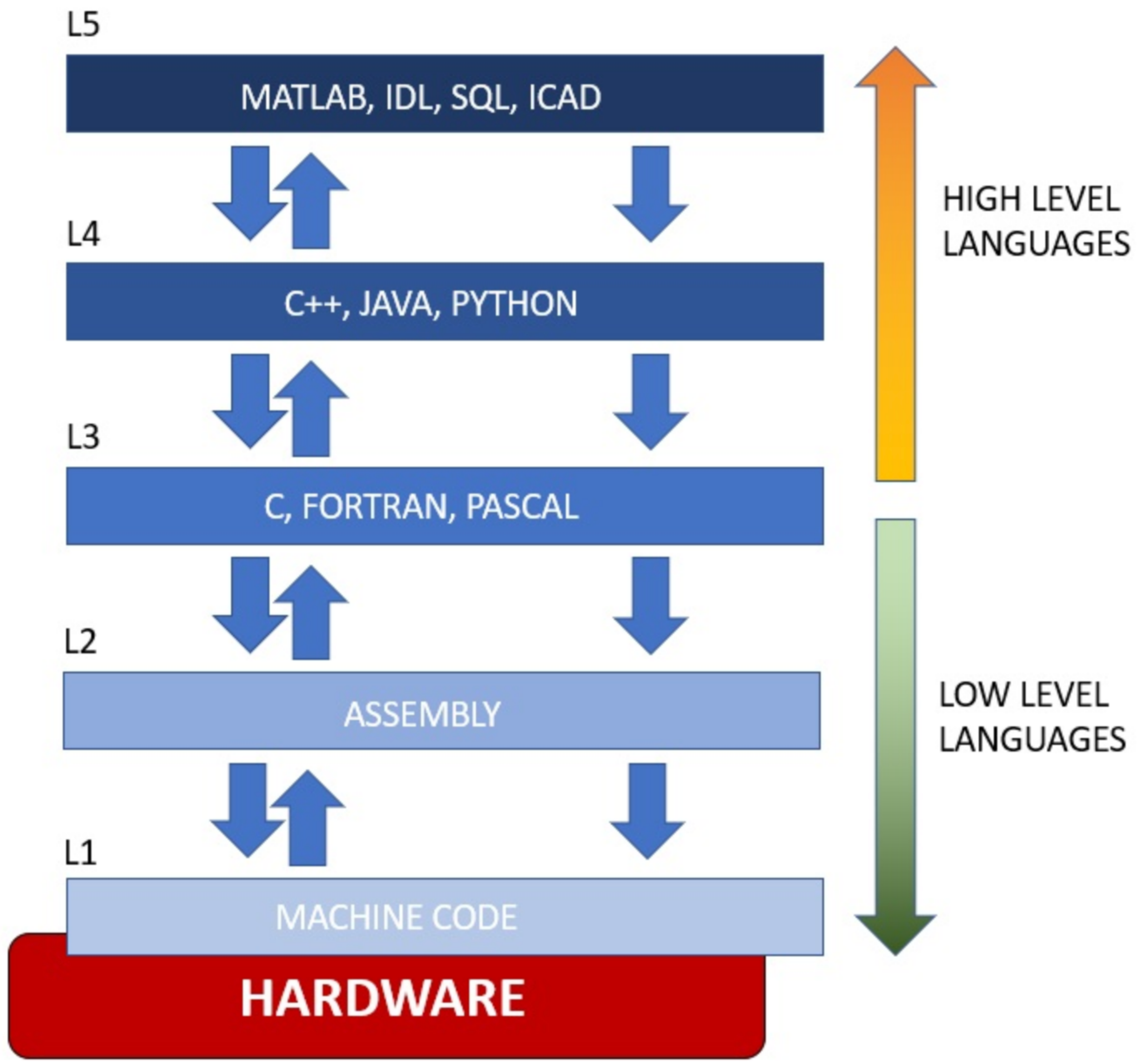
Also, we need to be explicit when declaring different types of variables.

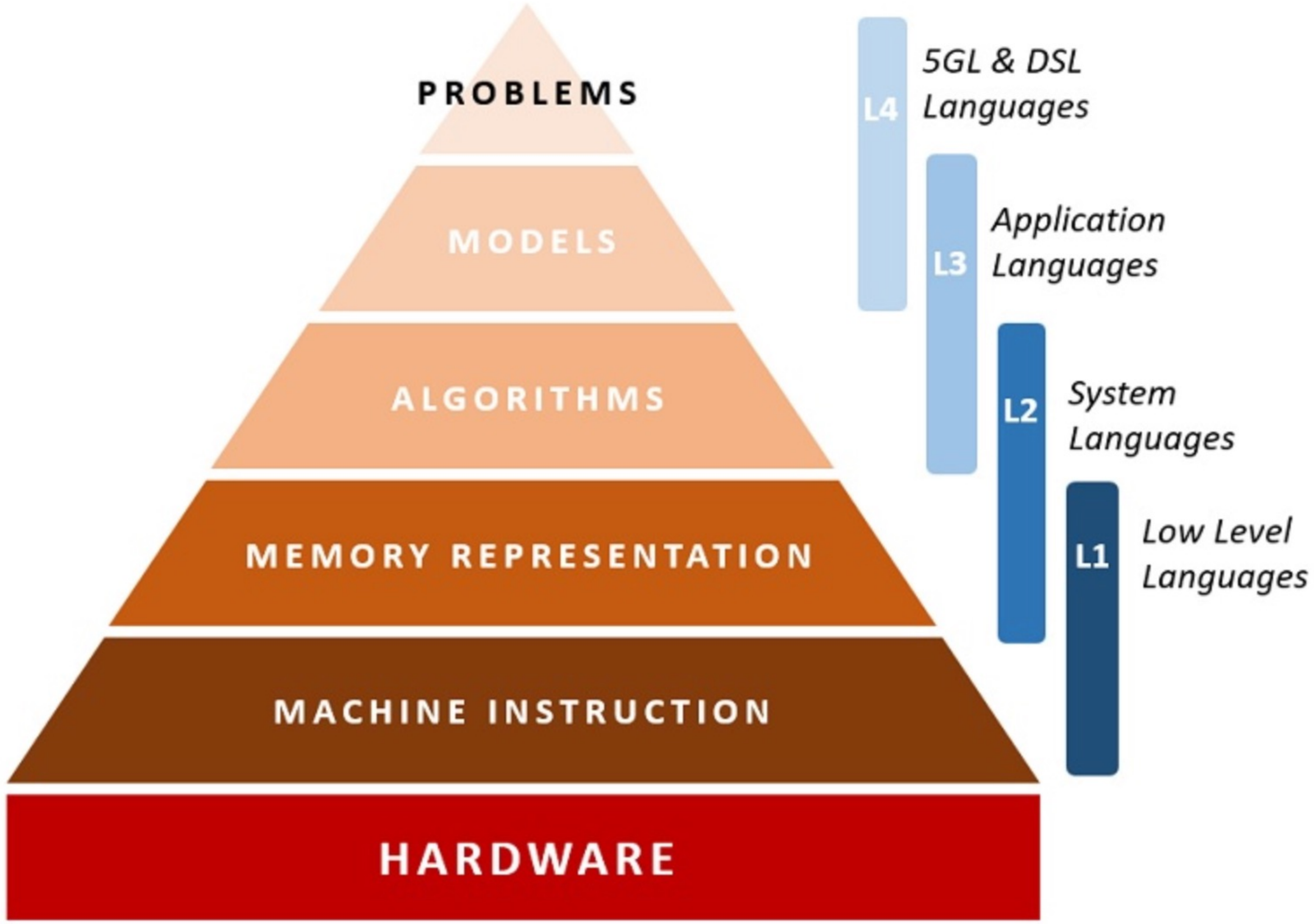
Both things not necessary in more modern languages.

## C

Although it is a bit redundant, it provides a better understanding of the inner working of software and how to better design them.

In fact, most of the modern programming languages were made using C. When we wish to create something larger than an application or an algorithm, such as a new system, it is better to use lower level languages (such as C), which provide better control on every component.


L5
MATLAB, IDL, SQL, ICAD
L4
C++, JAVA, PYTHON
L3
C, FORTRAN, PASCAL
L2
ASSEMBLY
L1
MACHINE CODE
HARDWARE
HIGH LEVEL LANGUAGES
LOW LEVEL LANGUAGES


PROBLEMS
MODELS
ALGORITHMS
MEMORY REPRESENTATION
MACHINE INSTRUCTION
HARDWARE
L4
5GL & DSL Languages
L3
Application Languages
L2
System Languages
L1
Low Level Languages

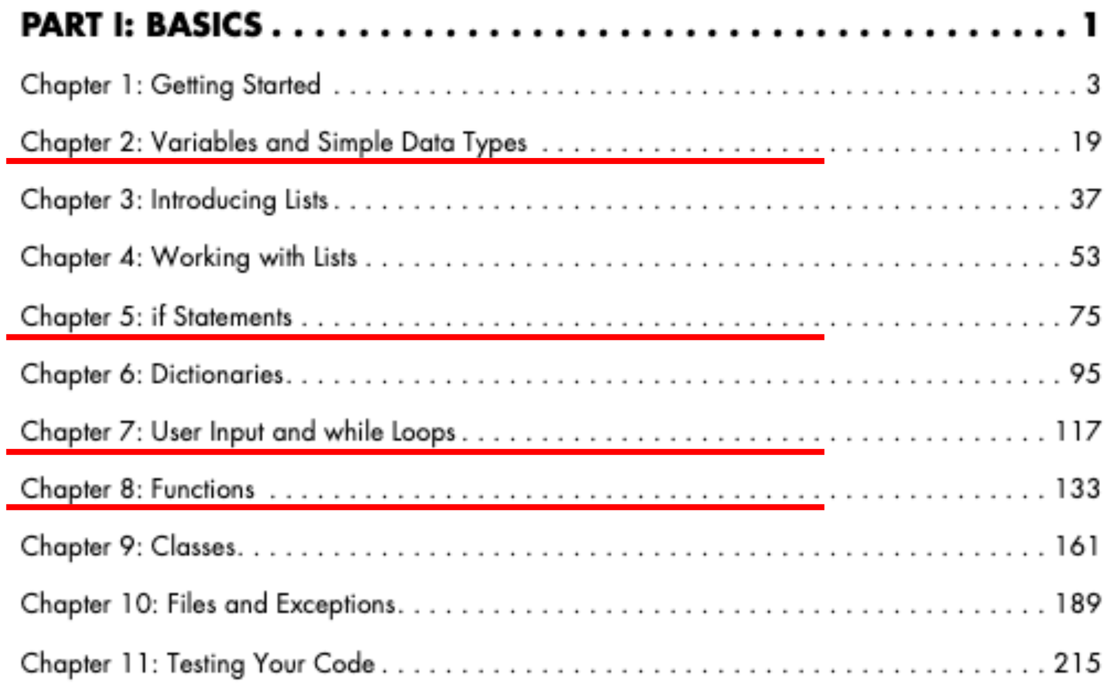
## Programming languages

There are a lot of common grounds between different programming languages.

In fact, the basic concepts of programming do not change between one language and another.

This is why knowing one language, greatly facilitates knowing another languages

## Python



## C