A brief introduction to CPU's and Programming Languages

What is a CPU? It is the central processing unit (or processor) of a computer. The CPU carries out the instructions of a program by performing control, input/output, arithmetic, and logical operations. It is the brains of your computer.

The CPU is located beneath the large fan and heat sink.

Here is a CPU with the heat sink and fan removed.

A CPU (central processing unit) implements an instruction set. An instruction set is the set of operations the CPU can perform. Some examples of instructions are

- Add
- Subtract
- Multiply
- Divide
- Compare
- Jump
- Increment
- Decrement

Some CPU's like Intel and AMD have compatible implementations of instruction sets

- the x86 instruction set was created by Intel based on the 8086 CPU and implemented by other CPU manufacturers like AMD, Cyrix, and VIA
- the x86_64 instruction set, which is a 64-bit extension of the x86 instruction set, was created by AMD and implemented by other companies like Intel and VIA

Other instruction sets like ARM, SPARC, Itanium, and PowerPC have different implementations and so x86/x86_64 applications will not run on them.

What is a programming language?

A special language which is used to write sets of instructions for computers to execute.

Programming languages typically are formalized, meaning

- they have defined structure and shape
- they have syntax (a set of rules for the words you put together)
- they have semantics (the meaning of the words you put together)

Low level programming languages:

- Machine language or machine code is the native language understood by the computer's CPU (central processing unit). This language is written using only 1's and 0's but people typically do not write programs using it.
- Assembly language is more readable and easier to write than machine code because it is written with human-recognizable instructions like
    - ADD, SUB, MUL, DIV
    - PUSH, POP
    - AND, OR, NOT
- C programming language. This was at one time considered a higher-level language but nowadays is considered a low level language. This is due to its lack of a runtime system (with memory management features like garbage collection) and its direct memory access capability.

Assembly and C code must be converted into an executable form in order to be run.

C is converted into Machine Code by a compiler (like GCC).

Assembly is converted into Machine Code by an assembler.

The output of both instances is Machine Code which can be understood by the CPU.

*NOTE: each of the CPU architectures require a compiler which has been written to produce Machine Code for its specific Instruction Set Architecture

Machine Code and Assembly Example

A program written in assembly language consists of a series of (mnemonic) processor instructions and meta-statements (known variously as directives, pseudo-instructions and pseudo-ops), comments and data. Assembly language instructions usually consist of an opcode mnemonic followed by a list of data, arguments or parameters. These are translated by an assembler into machine language instructions that can be loaded into memory and executed.

For example, the instruction below tells an x86/IA-32 processor to move an immediate 8-bit value into a register. The binary code for this instruction is 10110 followed by a 3-bit identifier for which register to use. The identifier for the *AL* register is 000, so the following machine code loads the *AL* register with the data 01100001.

```
10110000 01100001
```

This binary computer code can be made more human-readable by expressing it in hexadecimal as follows.

```
B0 61
```

Here, `B0` means 'Move a copy of the following value into *AL*', and `61` is a hexadecimal representation of the value 01100001, which is 97 in decimal. Assembly language for the 8086 family provides the mnemonic MOV (an abbreviation of *move*) for instructions such as this, so the machine code above can be written as follows in assembly language, complete with an explanatory comment if required, after the semicolon. This is much easier to read and to remember.

```
MOV AL, 61h        ; Load AL with 97 decimal (61 hex)
```

https://en.wikipedia.org/wiki/Assembly_language#Assembler

High level programming languages

- Python
- PHP
- Ruby
- Java
- others

The first three are Interpreted languages, meaning that they run within an interpreter and are interpreted line-by-line at runtime.

Java code is compiled into an intermediate representation called bytecode. This bytecode can then be run in an instance of the Java Virtual Machine (JVM).

For example:

contents of `HelloWorld.java`

```
//Hello World
class HelloWorld {
  public static void main(String[] args){
    System.out.println("Hello World!");
  }
}
```

The following command creates a `HelloWorld.class` file which contains the Java bytecode.

```
javac HelloWorld.java
```

To run this program we run it inside a JVM instance

```
java HelloWorld
```

Steps in compiling C programs

Contents of hw.c

```
/*
 * "Hello World!": A classic
 */

#include <stdio.h>

int main(void) {
      puts("Hello World!");
      return 0;
}
```

1. Precompile - replaces lines which begin with # with the contents they refer to

   `gcc –E hw.c > helloworld.c`

2. Compile - this step translates the precompiled (preprocessed) file into assembly instructions specific to the target processor architecture

   `gcc –S helloworld.c`

   `view helloworld.s`

3. Assembly - an assembler is used to translate the assembly instructions to machine code

   `gcc –c helloworld.c`

   To view the binary Machine Code

   `xxd –b helloworld.c`

4. Linking - The linker will arrange the object code as necessary and add other Machine Code containing the instructions for library functions used by the program. In the case of the “Hello, World!” program, the linker will add the object code for the puts function.

   `gcc –o helloworld helloworld.c`

   You can view the final Machine Code

   `xxd –b ./helloworld`

5. Now run the program

   `./helloworld`

https://www.calleerlandsson.com/the-four-stages-of-compiling-a-c-program/

Permission Bits

In Linux each file and directory has 3 user based permission groups. They are user, group, and other.

```
/home/glenn/pystuff> ls -l
total 20
-rwxrwxr-x. 1 glenn glenn  286 Jun  7 15:16 beer.py
-rw-rw-r--. 1 glenn glenn  146 Jun  7 16:21 ExampleProgram.java
-rwxrwxr-x. 1 glenn glenn 6480 Jun  8 11:13 helloworld
-rw-rw-r--. 1 glenn glenn  110 Jun  8 10:49 helloworld.c
```

Let's look at the ***file permissions*** on the `helloworld` program we compiled

```
-rwxrwxr-x. 1 glenn glenn 6480 Jun  8 11:13 helloworld
```

The first set of 3 letters `rwx` are the user (the owner of the file) permissions on this file.
The next set `rwx` are the group permissions.
The last set `r-x` are the other permissions.

```
r = read access on the file
w = write access on the file
x = execute access on the file
```

Now let's look at ***directory permissions***

```
ls -ld /home/glenn/pystuff

drwxrwxr-x. 2 glenn glenn 4096 Jun  8 11:24 /home/glenn/pystuff/
```

The `d` indicates that it is a directory

The first set of 3 letters rwx are the user permissions on this directory.
The next set rwx are the group permissions.
The last set r-x are the other permissions.

The letters have the following meanings for directories

`r` = the directory's contents can be shown
`w` = the directory's contents can be modified (create new files or folders; rename or delete existing files or folders); requires the execute permission to be also set, otherwise this permission has no effect
`x` = t he directory can be accessed with cd command; this is the only permission bit that in practice can be considered to be "inherited" from the ancestor directories, in fact if any folder in the path does not have the x bit set, the final file or folder cannot be accessed either, regardless of its permissions

https://wiki.archlinux.org/index.php/File_permissions_and_attributes

How to ***change permissions*** on u (owner), g (group), o (other), a (all three groups)

```
chmod [ugoa][-+][rwx] filename
```

Environment Variables

Environment variables in Linux systems are one of the ways you can modify and customize the way the system operates.

When you log into a system using a Desktop Environment and then open a terminal window you are creating a **non-login** shell. It is a non-login shell because you have already logged into the system. Non-login shells run the `~/.bashrc` file at startup. The tilde ~ is a shorthand way of expressing the path to your home directory.

When you open a terminal window you have a set of environment variables pre-set for you. These come from the `~/.bashrc` file. Modifying this file is one way you can customize your environment.

Example of a `~/.bashrc` file

```
# .bashrc

# Source global definitions
if [ -f /etc/bashrc ]; then
      . /etc/bashrc
fi

# User specific aliases and functions
PATH=$PATH:$HOME/bin

export PATH

# The PS1 variable is how you change your prompt
PS1='$PWD> '

# If I was running an Oracle database I would need something like this
ORACLE_SID=testdb

export PS1 ORACLE_SID
```

When you remotely connect into a system you are creating a **login shell** and the `~/.bash_profile` file gets run at startup. Modifying this file is another way you can customize your environment.

Example of a `~/.bash_profile` file

```
# .bash_profile

# Get the aliases and functions
if [ -f ~/.bashrc ]; then
      . ~/.bashrc
fi

# User specific environment and startup programs
```

Magic Numbers

Imbedded near the top of most files in Linux is a number signature which identifies the type of file it is. Some examples are

File Extension	Hex	ASCII
gz	1f8b	..
wav	5249 4646 a6fb 0100 5741 5645	RIFF....WAVE
jpg	ffd8 ffe0 0010 4a46 4946 0001	......JFIF..
Linux executable with no file extension	7f45 4c46	.ELF

Applications which expect files with certain filename extensions will use the magic number to validate that the file contents match the filename extension. For example a graphics program may expect files with extensions like `gif`, `jpg`, `tiff`, etc.

The system uses the magic numbers for files which don't have extensions, like compiled programs in the `/bin` or `/usr/bin` directories.

Magic numbers can be viewed like so

```
xxd /usr/share/pixmaps/faces/fish.jpg | head
```

or

```
hexdump /usr/share/pixmaps/faces/fish.jpg | head
```

or

```
vi /usr/share/pixmaps/faces/fish.jpg
```

`ESC` (if you are in insert mode)

```
:%!xxd
```

Differences between C and Python syntax

```
/*
 * Program to calculate the sum of 1,2,3...n
*/

#include <stdio.h>
int main()
{
    int num, count, sum = 0;

    printf("Enter a positive integer: ");
    scanf("%d", &num);

    // for loop terminates when n is less than count
    for(count = 1; count <= num; ++count)
    {
        sum += count;
    }
    printf("Sum = %d", sum);
    return 0;
}
```

```
#!/usr/bin/env python3.6

# 'main' is the name of the scope in which top-level code executes
edibles = ["ham", "spam","eggs","nuts"]
for food in edibles:
    if food == "spam":
        print("No more spam please!")
        break
    print("Great, delicious " + food)
else:
    print("I am so glad: No spam!")

print("Finally, I finished stuffing myself")
```

Notice that in Python:

- Control structures do not use Curly Braces, they use a colon: and proper indention.
- Variables are not pre-declared like they are in C.
- Lines do not end with a semi-colon like they do in C.
- The line `#!/usr/bin/env` is not a preprocessor directive like it would be in C.
- This line tells the shell to run the following code as a Python program. The shell looks inside the file for a Magic Number and doesn't see one but it does see `#!/usr/bin/env python3.6` and it knows how to run it. Note that `python3.6` is in my path. What would we need to do if it wasn't?
- See https://www.python.org/dev/peps/pep-0008/ for the Python coding style guide

Python Interactive Mode vs. Scripts

Way back in the day when you wanted to work on a computer you sat down at a keyboard and terminal which were physically plugged into a serial connection on the computer. If you visited a data center it is very possible that you would have seen a rolling cart with a CRT monitor and a keyboard on it. When you needed to connect to a physical console on a server you rolled the cart over to the rack and plugged the keyboard and monitor into the server, usually a 9-pin serial connection (RS-232).

Nowadays, you can just remotely login to a server with a program like SSH for example. When you do this the SSHD daemon running on the server will log you in and emulate a Terminal Device (called a Pseudo Terminal, PTS) for you and start your default shell for you (typically bash) .

Your bash shell has three input/output devices called stdin (Standard Input), stdout (Standard Output), and stderr (Standard Error). These devices are numbered 0, 1, and 2 respectively and when you are running bash in a Pseudo Terminal these file handles are attached to the PTS.

This means that bash receives its input from the keyboard, it prints its output messages to the terminal window, and it prints its error messages to the terminal window.

Some examples:

Redirecting stdout

`find ~` - find the files in my home directory and print them to stdout

`find ~ > ~/myfiles` - now find my files but redirect stdout to a file

`find ~ 1>~/myfiles` - same command as above, the 1 (stdin) is implicit in the command above

Redirecting stderr

`cat /etc/shadow` - this command prints a message to stderr because I'm not allowed to see this file

`cat /etc/shadow 2>/dev/null` - this command sends the message to a black hole. The 2 is not implicit, it must be explicitly stated on the command

Redirecting stdin

```
while read my_var
do
   echo "the file is $my_var"
done < ~/myfiles
```

The while-read command is a bash compound command. The main thing to notice is that we are redirecting the stdin of the read command to read from the file `~/myfiles`

Now when you run python from the command line like so

```
/home/glenn> python3.6
```

you are running it in interactive mode and its stdin, stdout, and stderr are attached to your pseudo terminal.

If you run a python script like so

```
/home/glenn> python3.6 beer.py
```

you are running it in script mode where the stdin is redirected to read from the file beer.py. When you are running python in script mode you cannot interact with the running program from the keyboard since the stdin for the python interpreter is redirected to read input from the script file.

Combining File Signatures, Permission Bits, and Redirection

If you have a python script with `#!/usr/bin/env python3.6` at the beginning of the file and it has its execute bit set, when you run it, it runs as a python program with its stdin redirected to read input from the script file.

```
/home/glenn/pystuff> ls -l beer.py
-rwxrwxr-x. 1 glenn glenn 286 Jun  7 15:16 beer.py

/home/glenn/pystuff> cat beer.py
```

```
#!/usr/bin/env python3.6

def sing(b, end):
    print(b or 'No more','bottle'+('s' if b-1 else ''), end='')

for i in range(99, 0, -1):
    sing(i, 'of beer on the wall,')
    sing(i, 'of beer,')
    print('Take one down, pass it around,')
    sing(i-1, 'of beer on the wall.\n')
```

https://github.com/Alver23/CursoPython/blob/master/Clase2/beers.py

The code above shows some of the commands which are different in Python 3 vs. Python 2.
The print statement now is `print()` instead of `print`, and to print text and not wrap you use `print('some string', end='')` instead of `print 'some string',`
Another change is the command is now `range()` instead of `xrange()`. See these articles for more details http://sebastianraschka.com/Articles/2014_python_2_3_key_diff.html
https://docs.python.org/3/whatsnew/3.0.html

Python Fundamentals

Data Types:

Strings:

- a string is an ordered sequence of characters enclosed in single or double quotes
- strings are zero-indexed (0, 1, 2, 3 … n)
- strings are immutable
- strings have many useful methods

Strings are indexed

```
>>> good_career = 'Cyber Security'
>>> good_career[6]
'S'
```

Strings are immutable

```
>>> good_career[6] = 's'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment

Strings have length
>>> len(good_career)
14
```

Strings can be sliced. The operator `[m:n]` returns part of the string starting at m and up to but not including `n`

```
>>> good_career[0:6]
'Cyber '
>>> good_career[0:5]
'Cyber'
>>> good_career[:5]
'Cyber'
>>> good_career[6:14]
'Security'
>>> good_career[6:]
'Security'
```

String Methods

```
>>> good_career.upper()
'CYBER SECURITY'
>>> good_career.lower()
'cyber security'
>>> good_career.split()
['Cyber', 'Security']

>>> '   spacious   '.strip()
'spacious'
>>> 'www.example.com'.strip('cmowz.')
'example'

>>> my_number = '3456'
>>> my_number.isdigit()
True

Strings can be compared
>>> my_number == '2345'
False
>>> my_number == '3456'
True
```

Tuples:

- a ordered series of values (comma sep. if more than one) optionally surrounded by parentheses
- tuples are zero-indexed
- Like a row in a database table (also called a tuple).
- Often different data types are packed into a tuple.
- Tuples are immutable. Once they are created they can't be changed
- Tuples can be unpacked
- One element tuples must be created with a final comma

Creating a tuple and indexing its values (indices start at 0)

```
>>> my_tup = ('AA', 32, 'Mon')
>>> my_tup[0]
'AA'
>>> my_tup[1]
32
```

Creating another tuple

```
>>> your_tup = ('BB', 44, 'Tue')
```

Adding tuples together

```
>>> our_tup = my_tup + your_tup
>>> our_tup
('AA', 32, 'Mon', 'BB', 44, 'Tue')
>>> our_tup[3]
'BB'
>>>
>>> type(our_tup)
<class 'tuple'>
```

Unpacking a tuple

```
>>> a1, a2, a3, b1, b2, b3 = our_tup
>>> a1
'AA'
>>> b3
'Tue'
```

Doing math with tuple elements

```
>>> our_tup[1] * our_tup[4]
1408
```

Length of a tuple

```
>>> len(our_tup)
6
```

Tuples are immutable – they cannot be changed

```
>>> my_tup[0] = 'CC'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

One element tuple

```
>>> just_one = '1',
>>> type(just_one)
<class 'tuple'>
>>> just_one = '1'
>>> type(just_one)
<class 'str'>
```

Lists:

- an ordered series of values (comma sep. if more than one) surrounded by square brackets
- lists are zero-indexed
- lists can be composed of mixed data types
- lists are mutable. Unlike a tuple, the values in a list can be changed

Create a list

```
>>> my_list = [ 'aa', 'bb', 'cc', 'dd']
>>> my_list
['aa', 'bb', 'cc', 'dd']
```

Add to a list

```
>>> my_list.append('ee')
>>> my_list
['aa', 'bb', 'cc', 'dd', 'ee']
>>> your_list = my_list + ['ff']
```

Delete a value from a list

```
>>> my_list.remove('ee')
>>> my_list
['aa', 'bb', 'cc', 'dd']
```

Delete an element from a list

```
>>> del my_list[3]
>>> my_list
['aa', 'bb', 'cc']
```

Pop an element from a list

```
>>> pv = my_list.pop(2)
>>> pv
'cc'
```

Lists can be sliced

```
>>> num_list = [3, 4, 5, 6, 7, 8]
>>> num_list[0:3]
[3, 4, 5]
>>> num_list[3:]
[6, 7, 8]
```

Lists can be sorted

```
>>> random_letters = ['Q', 'b', 'x', 'S', 'M', 'j']
>>> random_letters.sort()
>>> random_letters
['M', 'Q', 'S', 'b', 'j', 'x']
```

Sets:

- a series of values (comma sep. if more than one) surrounded by curly braces
- sets do not allow duplicate values
- sets are unordered and do not support indexing
- sets are good for de-duplicating values
- sets are good for membership testing

Create a set with duplicate values, the duplicates get removed

```
>>> unique_set = {'aa', 'bb', 'cc', 'dd', 'aa', 'bb'}
>>> unique_set
{'dd', 'cc', 'bb', 'aa'}
```

Sets have length

```
>>> len(unique_set)
4
```

Sets are unordered, and are not indexed

```
>>> vv = unique_set[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'set' object does not support indexing
```

Convert a list with duplicates into a set without duplicates

```
>>> names = ['mark', 'steve', 'joe', 'bill', 'steve', 'margo', 'joe',
'larry']
>>> unique_names = set(names)
>>> unique_names
{'bill', 'margo', 'larry', 'mark', 'joe', 'steve'}
```

Test if a specific value is in a set

```
>>> 'fred' in unique_names
False
>>> 'margo' in unique_names
True
```

Dictionaries:

- a series of unordered key:value pairs (comma sep. if more than one) surrounded by curly braces
- dictionaries do not allow duplicate keys, the most recent key:value is retained, the other is dropped
- the key values in dictionaries are not limited to integers

Create a dictionary and add to it

```
>>> santa = { 'fname':'Kris', 'lname':'Kringle' }
>>> santa
{'fname': 'Kris', 'lname': 'Kringle'}
>>> santa['address'] = 'North Pole'
>>> santa
{'fname': 'Kris', 'lname': 'Kringle', 'address': 'North Pole'}
>>> santa['age'] = 'really old'
>>> santa
{'fname': 'Kris', 'lname': 'Kringle', 'address': 'North Pole', 'age':
'really old'}
```

Dictionaries can be composed of other types

```
>>> hobbies = {'mine':('reading','hiking','eating'),
'spouse':('cooking','jogging','sewing')}
>>> hobbies['mine']
('reading', 'hiking', 'eating')

>>> my_stocks = {
... 'IBM':{'current price':32.44, 'low price': 30.11, 'high price':
35.09},
... 'WHH':{'current price':11.38, 'low price': 11.01, 'high price':
25.88}
... }
>>> my_stocks
{'IBM': {'current price': 32.44, 'low price': 30.11, 'high price':
35.09}, 'WHH': {'current price': 11.38, 'low price': 11.01, 'high
price': 25.88}}
>>> my_stocks['IBM']
{'current price': 32.44, 'low price': 30.11, 'high price': 35.09}
```

Remember:

```
Strings         - immutable
Sets            - immutable
Tuples          - immutable
Lists           - mutable
Dictionaries    - mutable
```

Input

```
>>> inp = input('Please enter your name: ')
Please enter your name: Security Steve
>>> inp
'Security Steve'
>>> print('your name is: ' + inp)
your name is: Security Steve
```

Printing

```
>>> stock = 'IBM'
>>> shares = 100
>>> price = 23.99
>>> print(stock, shares, price)
IBM 100 23.99
>>> print('Stock: %10s Number of Shares: %10d Price: %10.2f' % (stock,
shares, price))
Stock:        IBM Number of Shares:        100 Price:      23.99
>>> print('Stock: %-10s Number of Shares: %-10d Price: %-10.2f' %
(stock, shares, price))
Stock: IBM        Number of Shares: 100        Price: 23.99
>>> print('Stock: %s Number of Shares: %d Price: %.2f' % (stock,
shares, price))
Stock: IBM Number of Shares: 100 Price: 23.99

>>> print('Stock: %s Shares: %d Price: %.2f' % (stock, shares, price))
Stock: IBM Shares: 100 Price: 23.99
>>> print('Stock: {:>s} Shares: {:>d} Price: {:>.2f}'.format(stock,
shares, price))
Stock: IBM Shares: 100 Price: 23.99
```

References:

https://en.wikipedia.org
https://www.techwalla.com/articles/types-of-computer-languages
https://usercontent2.hubstatic.com/13292679_f520.jpg
http://cdn.mos.cms.futurecdn.net/a450b98316e9cc1bd4ee712e9e0b08e6-320-80.jpg
https://tiswww.case.edu/php/chet/bash/bashref.html
https://www.calleerlandsson.com/the-four-stages-of-compiling-a-c-program/
https://wiki.archlinux.org/index.php/File_permissions_and_attributes
http://www.python-course.eu/python3_for_loop.php
https://www.programiz.com/c-programming/c-for-loop
http://greenteapress.com/wp/think-python-2e/
https://docs.python.org/3/index.html
https://www.python.org/shell/