Computer Science 1: Notes

Table of Contents

1. Preface

This is all you need to ace the upcoming computer test, if you spend 30 minutes to 1 hour reading this document There's a high chance you'll get a 20/20, but there's also no need to read all of it, I would recommend going through the "Table of Contents" and just looking at the things you don't understand the most, and revising them. any questions about any content can be asked in the class group, but not in private. This paper doesn't have any additional information that wasn't taken in class that will appear in the test. I would recommend reading the Exam studying methodology 5 after this paragraph.

2. Computer Architecture

2.1. Introduction

2.2. CPU

- The CPU (**C**entral-**P**rocessing-**U**nit) is the square component we saw in class inside the computer.
- It consists of many smaller units that handle different things, but in the explanation we will take, it will only contain an ALU 2.2.1 and a CU 2.2.2, both of their explanations will be simplified.
- It's essentially the brain of the computer, handling all the computation (we can call them tasks) that happen on the computer.

Figure 1: A physical CPU

,

2.2.1. ALU

- The ALU (**A**rithmetic-**L**ogic-**U**nit) is a component/unit of the CPU 2.2
- It handles all the arithmetic (addition, subtraction, division, multiplication, etc...)
- All the operations that happen inside the ALU are in base-2

2.2.2. CU

- The CU (**C**ontrol-**U**nit) is a component/unit of the CPU 2.2
- It will manage and execute instructions.
- For now, we will think of instructions as operations needed to perform certain actions on the computer (adding two numbers, opening an app, etc...)

2.2.3. CPU Characteristics

- A CPU has a clock speed: the speed of which instructions will execute, and it is a frequency.
- $1\text{GHz} = 1 * 10^9$ Hz

2.3. Memory

- Memory in the context of a computer system is the place where "data" will be stored, data can be for example: images, text, files, Variables 3.3, etc...
- We will talk about two types of memory: Primary Memory, and Secondary Memory.
- We will talk about the data units: bit, byte, kb, mb, gb.
- **Primary Memory**: Characteristics of primary memory is that it's fast, and "volatile", but what does volatile mean? In this context, it will mean that after the computer restarts, the data will not be saved, for example: when playing a game, the amount of health you have will not be saved after the computer restarts, it only ever was in-memory the time you were playing before. Examples of primary memory is RAM 2.4.
- **Secondary Memory**: Secondary memory is slow, but non-volatile, meaning the data will be saved after the computer restarts, this would be ideal for images for example, you would not want them to dissapear after you restart.

2.4. RAM

- RAM (**R**andom-**A**ccess-**M**emory) is the a type of memory.
  It's the green sticks I've shown you in class.
- RAM is primary memory.

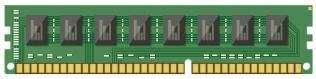

Figure 2: A physical stick of RAM

,

2.5. HDD

- HDD (Hard-Disk-Drive) is a type of memory. It's the metal square Ive shown you in class.
- HDD is secondary-memory.

Figure 3: An HDD

2.6. Converting data sizes (In the decimal system (SI))

- The bit is the single smallest data unit.
- 1 byte $= 8$ bits
- 1 kilobyte $= 1000$ bytes
- 1 megabyte $= 1000$ kilobytes
- 1 gigabyte $= 1000$ megabytes

2.7. Number Systems

2.7.1. Base-10

- Base-10 is the number system we humans use.
- It's made up of 10 digits, digits from (0 to 9): 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.
- Any other number is made up from these the 9 digits.

2.7.2. Base-2

- Base-2 is the number system is computer uses.
- It's made up of two digits: `0` and `1`.
- Any other number is made up from these thee `2` digits.

2.7.3. Relationship between base-2 and base-10

Numbers can be convertered to-and-back from base-2 to base-10.

2.7.3.1. Converting from Base-10 to Base-2

We will learn the proccess by trying to solve for x in: $x_2 = (75)_{10}$, so we need to convert 75 to the base-2 system.

- Step 1: Make a table.
- Step 2: Divide `75` by `2`.
- Step 3: If the number is a decimal, take everything before the decimal and assign it the bit `1`, if the number is a whole number, write the number as is and assign it the bit `0`.
- `(75/2) = 37.5`, we will take `37` and set the bit to `1`, then we repeat until `1`, `(37/2) = 18.5`, we take `18`, then `18/2 = 9`, we take `8` and set the bit to `0`, then `9/2 = 4.5`, we take `4` and set the bit to `1`, `4/2 = 2`, and `2/2 = 1`, then `1/2 = 0.5`(special scenario) and set the bit to `1` and stop there..
- Step 4: Construct the number, we will take the bits from the table bottom-up, so:

number	index	bit
37	0	1
18	1	1
9	2	0
4	3	1
2	4	0
1	5	0
$\frac{1}{2}$	6	1

Table 1: Table for converting 75 to base-2.

$(75)_{10} = (1001011)_2$

2.7.3.2. Converting from Base-2 to Base-10

We will take the same example as above and in Table 1, and we will use the indexes to remake the base-10 number.

we had $(75)_{10} = (1001011)_2$

We need to only consider the "1"s and their indexes as shown in Figure 4

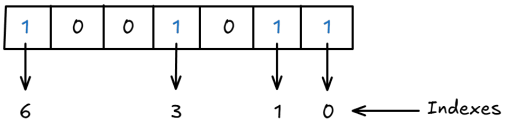

Figure 4: The 1's and their indexes.

$(1001011)_2 = (2^0 + 2^1 + 2^3 + 2^6)_{10} = (75)_{10}$

3. Python

3.1. Introduction

Definition: Python is a "high-level", "interpreted", "general-purpose" scripting language.

Explanation of the three characteristics:

- **general-purpose**: A programming language that can be used for most use-cases and does not follow one paradigm.
- **interpreted**: A programming language is interpreted when there is an external program (the "interpreter") that is reading the instructions line by line and executing the instructions that would otherwise not be executable by the CPU (Figure 5). By contrast, a compiled programming language is a language that gets translated to instructions the CPU can execute directly by the "compiler" (Figure 6).

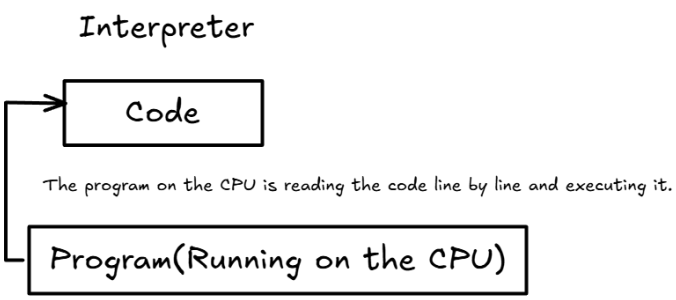

Figure 5: Interpreted languages process.

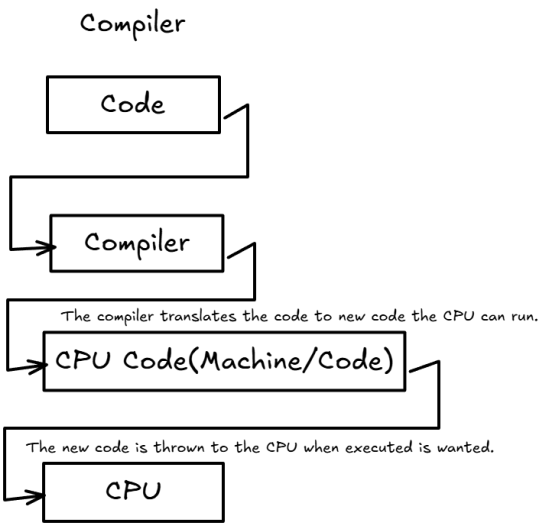

Figure 6: Compiled languages process.

- **high-level**:

  Built on many lower-level "abstractions", "abstractions" in this context are harder programming languages, concepts and technologies that Python builds upon to provide an easy language.

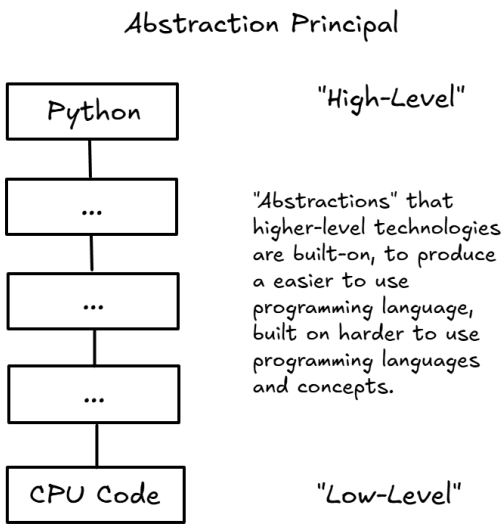

Figure 7: Abstractions.

Additional characteristics:

- **sequential**: Python executes instructions by sequence by default, from top to bottom.
- **imperative**: The programming style Python uses to maintain state.

3.2. Data Types

- **int**: (integer) a whole number, example: `10`, `25`, `300`; numbers with a decimal place are not integers(`10.5`, `2.8`, `1.0`).
- **float**: A decimal number, example: `52.6`, `19.3`; numbers with `0` as a decimal place are floats, like: `1.0`, `5.0`.
- **bool**: A data type which is either `True`, or `False`.
- **str**: A sequence of characters, example: `"Hello"`, `"This is a sentence"`, `""`.
- **list**: A sequence of elements.

example:

```
grades = [10, 5.2, 9.8, 12.5, 13.2]
names = ["John", "Johnny", "Johnnathan", "John"]
empty = []
list_of_lists = [["element 1", "element 2", "element 3"], [5, 10, 20]]
```

Characteristics:

- Lists are heterogenous in Python: meaning lists can hold multiple data types and are not forced to maintain one data-type, a list can have elements of any other data-type.
- The length of a list is how many elements it holds, for example, the length of `grades` is `5`, and of `empty` is `0`.
- The index of an element in a list is the position of that element `0`-based, and can be used to access that element, example:

`grades[0]` would give `10`, `grades[1]` would give `5.2`, and so on...

- List indexing is non-commutative: $\lambda\upsilon \neq \upsilon\lambda$

You cannot access lists by writing the index before the list, meaning: 0[grades] is not valid. Only grades[0] is valid.

- **dict**: (dictionary) is a key-value store, example:

```
person = {"name": "John", "age": 30}
```

This is a dict that has two keys: `name` and `age`, with values: `"John"` and `30`.

- `D[K]` where `K` is the key, is the value of that key in that dict, example: `person["name"]` would give us "John".

3.3. Variables

Variables in Python have two characteristics, they are "mutable" and "untyped".

- "mutable": They can change, evolve and be re-assigned.
- "untyped": They do not have a specific type.

examples:

```
name = "Johnny"
johnny_age = 20
grades = [10, 5.2, 9.8, 12.5, 13.2]
person = {"name": "John", "age": johnny_age}
```

Here, we have declared four variables, and assigned each to their own value. Note that `person["age"]` would be `20`, since it's using the variable `johnny_age`.

3.4. Operators

3.4.1. Arithmetic operators

- 1.) Addition (+)
- 2.) Subtraction (-)
- 3.) Multiplication (*)
- 4.) Division (/)

Note: int $\in \mathbb{Z}$

- $\text{int} + \text{int} \rightarrow \text{int}$
- $\text{int} - \text{int} \rightarrow \text{int}$
- $\text{int} * \text{int} \rightarrow \text{int}$
- $\frac{\text{int}}{\text{int}} \rightarrow \text{float}$

Why is $\frac{\text{int}}{\text{int}} \rightarrow \text{float}$ and the others don't behave this way? Because there isn't any two integers you can add to each other that will give you a decimal number, but there are integers that can divide each other to give you a float, so a `int` divided by an `int` will always result in a float. Other type theoretics include:

- $\text{str} + \text{str} \rightarrow \text{str}$

- float + float → float
- int + float → float

3.4.2. Comparison operators

We will define variables for example-purposes:

```
first_number = 10
second_number = 5
first_string = "John"
second_string = "Johhny"
```

- 5.) Equality (==): Will return a `bool` to see whether the two sides are equal. So we can say:

$\forall(x, y) \Rightarrow (x == y) \rightarrow \text{bool}$

Translation(haha): For every variable `x` and `y`, `x == y` will always return a `bool`.

examples:

- `first_number == first_number` would be `True`, because `10 == 10`, and also, $\forall(x) \Rightarrow (x == x) = \text{True}$ (For every variable `x`, `x == x` will always be `True`).
- `first_string == second_string` would be `False`, because `John` is not equal to `Johnny`.
- 6.) Inequality (`!=`): Will return a `bool` to see whether the two sides are not equal. It holds the same type-theory as equalities.

examples:

- `first_number != first_number` would be `False`, because `10` is equal to `10`.
- `second_string != first_string` would be `True`, because `John` is not equal to `Johnny`.

3.4.3. Logical operators

- 7.) Negation (`not`): Given a `bool` it will return the inverse of that `bool`. By definition, if variable `x` is a `bool`:

$\forall(x) \Rightarrow (\text{not } x) \rightarrow \text{bool}$

Translation: For any variable `x`, `not x` will give a `bool`.

Alternatively, we might see `not` as $\neg$.

example:

- `not True` = `False`.
- `not False` = `True`.
- 8.) Bigger-than, less-than, bigger-than-or-equal, less-than-or-equal (<, >, <= >=): Will return a `bool`.

examples:

- `10 > 5` = `True`.
- `10 < 5` = `False`.

3.4.4. Bitshift operators

- 9 .) Left-bitshift (<<): `x << n`, where `x` and `n` are `ints`, will shift the bits of `x` by `n` zeroes and return the base-10 `int`.

Example:

- `5 << 2`:
- If we convert 5 to base-2: $(5)_{10} = (101)_2$.

- And left shift by 2: `(10100)` (We added two zeroes.)
- Now if we convert back to base-10, the number becomes: $2^2 + 2^4 = 20$.
- 10 .) Right-bitshift (>>): The inverse of left-bitshift.

Example:
- `5 >> 2`
- If we convert 5 to base-2: $(5)_{10} = (101)_2$.
- And right shift by 2: `(001)`, (The `01` that was there moved to somewhere that doesn't exist.)
- Now we convert back to base-10, the number becomes: $2^0 = 1$.

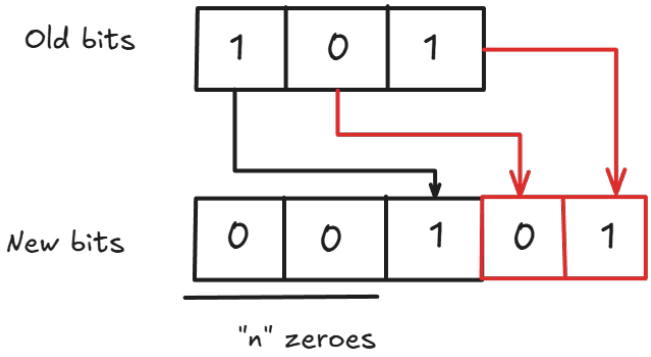

Figure 8: Right bitshift

3.4.5. Operation + Assignment

All the arithmetic, and bitshift operators have an assignment equivelant used to modify variables.

examples: We will take those variables for the examples:

```
first_number = 10
second_number = 5
first_string = "John"
second_string = "Johhny"
```

Applications:

```
first_string += " Jason" # first_string will be "John Jason" after that, because "John" + "
Jason" = "John Jason".
second_number -= 6 # second_number will be -1 after that, because 5 - 6 = -1.
second_number *= -2 # second_number will be 2 after that, because (-1)(-2) = 2.
first_number /= 5 # first_number will be 2.0 after that, because 10 / 5 = 2.0.
first_number = 5 # first_number will be 5 after that, normal assignment.
first_number <<= 2 # first_number will be 20 after that, because 5 << 2 = 20
first_number >>= 3 # first_number will be 2 after that, because 20 >> 3 is 2.
```

3.5. Built-in functions

- `print(m)`: `m` can be anything of any data-type, the print function will display `m` to the IO,

example: `print("Hello")`, `print(10)`.
- `type(obj)`: obj can be any data-type, type will return the type of `obj`.

example: `type("Hello")` will be `str`. `type(10.0)` will be `float`.
- `int(x)`: Is a function that will return the integer-version of what it passed to it.

example:
- `int(10.0)` will be `10`.
- `int("2")` will be `2`.
- `int(5)` will still be `5`.

3.6. Functions

We will imagine functions as abstractions 7 that take inputs, and result in outputs.

3.6.1. A quick visual before jumping in

```
def add(x, y):
  return x + y
```

Code 1: Function that adds two inputs together

3.6.2. Function signature & body

3.6.3. Function inputs & outputs

3.6.4. Function parameters & arguments

3.6.5. Pure & deterministic functions

3.7. Classes

3.7.1. Members

3.7.2. Methods

3.7.3. Dunder methods

3.7.4. Inheritance & Polymorphism

3.7.5. Encapsulation & Abstraction

3.7.6. Generator & decorator functions

3.8. Try it yourself

You can go to the website: https://www.programiz.com/python-programming/online-compiler/, and follow everything each step of the way, maybe even ask AI but Ill be pretty upset if I found out :(

4. Miscellaneous

4.1. Cache

A cache is a place of storage of temporary information that get added(**inserted**) and removed (**evicted**) based on certain criteria, the criteria to add information in the cache is called an **insertion policy**, and the word to remove something from the cache is called an **eviction policity**, this mechanism is mainly used for performance reasons, for

- **Tiktok example scenario** : A Tiktok video is stored in the US, and you amongst other Lebanese people watch that video really often on their phones, Tiktok might consider that fact, and decide to insert the video in a cache (the placement of storage) that is closer to Lebanon, that way when people want to go watch that video, it goes and gets that video information from somewhere closer, resulting in faster speeds.

4.1.1. The client-server model

The client server model is made up of two components, the: the client, and the server, and focuses on the relationship and communication that happens between these two components.

4.1.1.1. The client

The client (also the consumer in this relationship), will consume information, this can be for example: you watching Tiktok on your phone, the client is your phone.

4.1.1.2. The server

The server (the producer in this relationship), will produce and send information to the client, in the example of Tiktok, the Tiktok **servers** will send out the video information (title, description, comments, likes, reposts, video buffer, etc...) to your phone.

4.1.1.3. Illustration

Figure 10: The client-server model

4.1.1.4. The caching of client-server models

Usually caching in the client-server model is based on geographical location, for this, you can imagine tiktok example scenario 9, let's first look at the figure Figure 11

Figure 11: Cache added on a standard client-server-model

Here, the Tiktok system notices that multiple people, close to each other geographically, are watching the same video very frequently, so by nature, to satisfy those people's need for performance, the system decided to implement a cache, so instead of every time having to get the video information from New York, America, the information is gathered from Paris, France, resulting in a shorter resposne time, naturally data is only needed to be cached once, and is only cached again if it is evicted, so for example, after some amount of time, the system might decide: "There's no one watching this, let me free up space and free it from the cache."

4.1.2. In the context of a CPU

We can also apply the idea of a cache inside a CPU, we have said before that a Compiler 6 turns normal code into CPU code (more commonly called machine code), this machine code will have to actually **consume** this code to do something meaningful with it, so a execution process without a cache might look like Figure 12, but with a cache implemented, might be like Figure 13, internally.

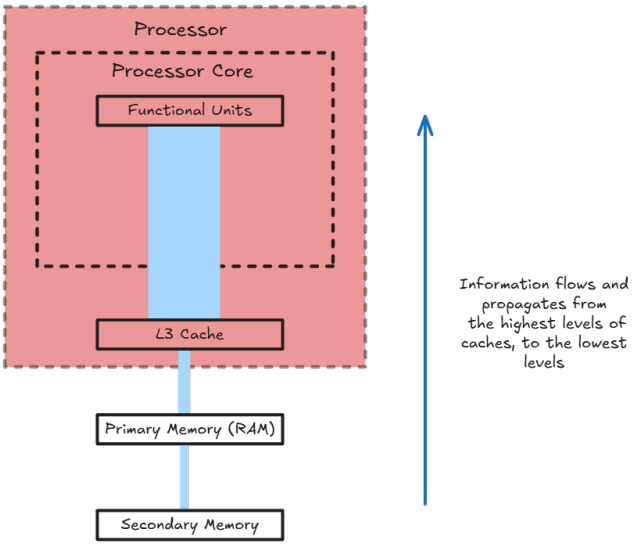

Figure 12: CPU without cache

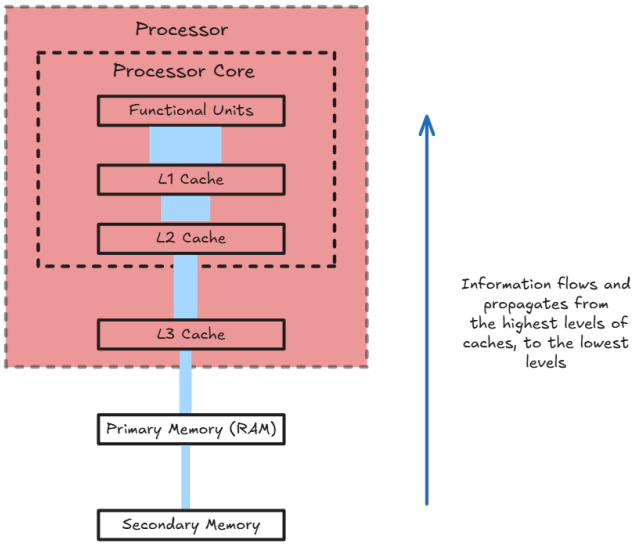

Figure 13: CPU without cache

4.1.2.1. CPU cache levels

In modern systems, usually CPUs have 3 cache levels, L1 cache, L2 cache and L3 cache, when the CPU can't find information in the L1 cache, it will go to the L2, if it can't find in the L2 it will go to L3, if it's not in the L3 it will go to primary memory, if it's not in primary memory it will go to secondary memory. This is called **propagation**.

In resume, CPU caches improve performance, but add complexity on an existing system.

4.1.3. Summary

A cache is an Abstraction Layer 7 that is usually implemented between a producer and a consumer, a sender and a receiver, and emitter and a subscriber, that stores information temporary in a way that can be accessed faster for that consumer, such that there are performance gains.

5. Exam studying methodology

- Focus on understanding the concepts before memorizing them, there's not even the need to memorize them.
- 20% of the event yields 80% of the outcome, don't try and nail all the details, just focus on the most important things.
- Focus on the English parts of things and try to explain thing to yourself and other people to see if you understand.
- Regarding Python, you can try everything each step of the way, read Try it yourself 3.8
- Ask me in the group if there's any questions, not in private, I won't respond.

Good Luck! I want good grades.

Author: Michael Salloum