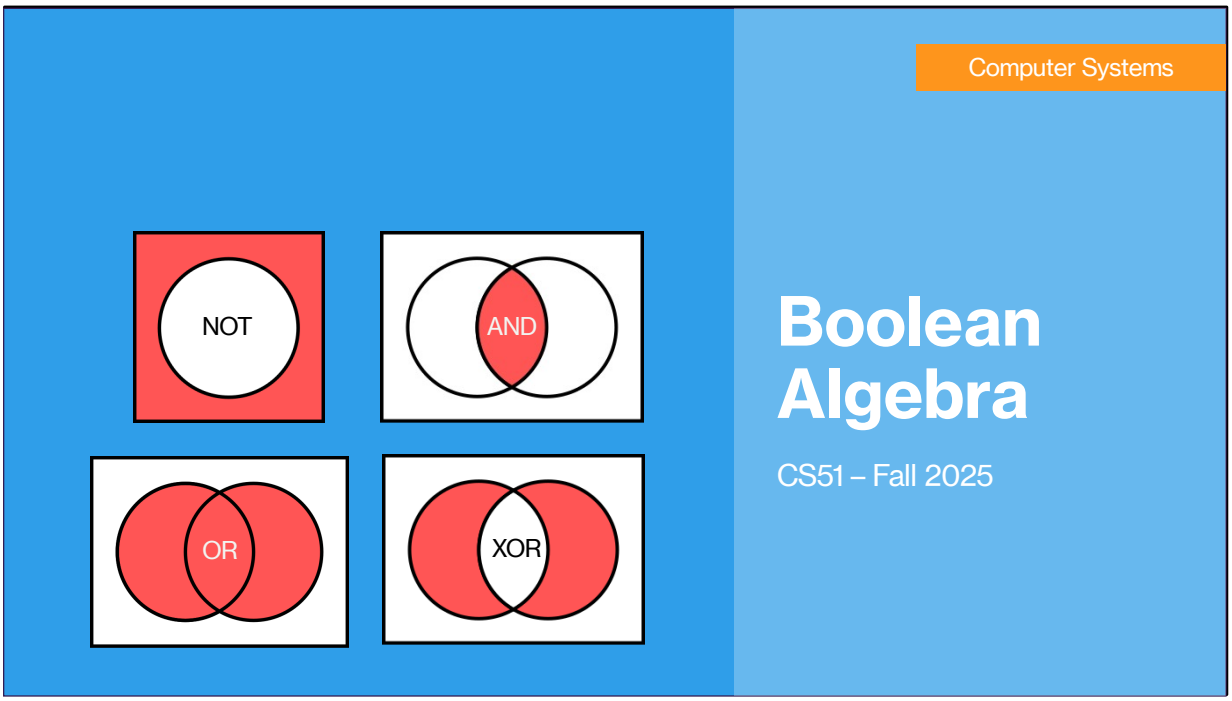


Computer Systems

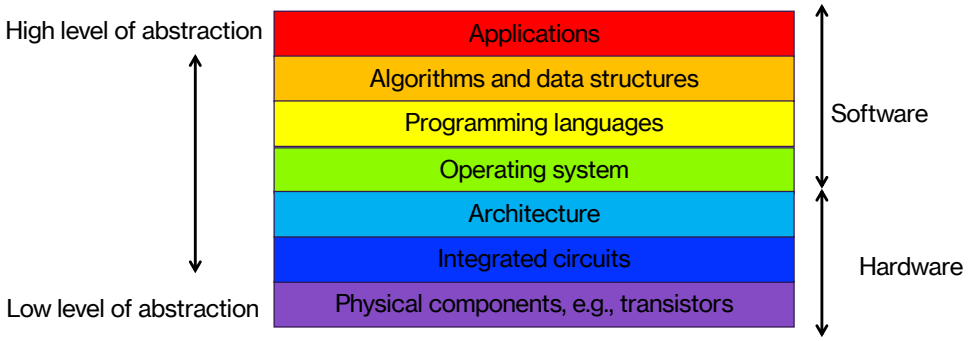
Boolean Algebra

CS51 – Fall 2025

Today will commence a new unit in our class that will focus on computer systems. We will see how major computer chips are made from physical components like transistors that control two states and thus give rise to a binary representation that enables us to work with a branch of mathematics called Boolean Algebra. We will talk about this concepts both in terms of applying logical operations on abstract 0s and 1s and of how they appear in an advanced programming language like Python.

Abstraction

- Critical technique for managing complexity. We will simplify and generalize information by hiding details when they are not important. This allows us to cultivate a higher level of understanding without being overwhelmed with insignificant details.



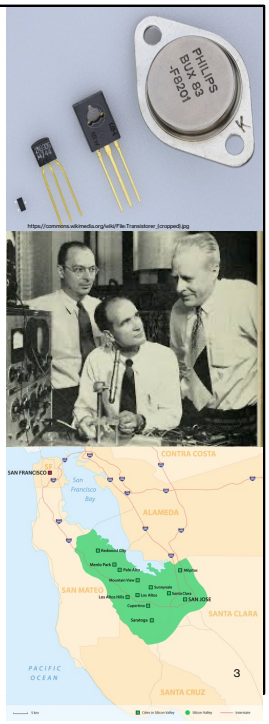
2

It can get pretty easy to get lost in the details of how computers are built and work. So how do we go about it? One of the characteristics that distinguishes computer scientists is a systematic approach to managing complexity by using abstraction. The figure above illustrates levels of abstraction for a computer system. At the highest level of abstraction, we have all applications that our computers offer and users interact with. Think of your favorite browser, music player, photo and video viewing applications, word and spreadsheet processors, etc. Programmers write these applications in code by using algorithms and data structures which they implement in different programming languages, such as Python, Java, Haskell, etc, based on their specific needs. One lower level within the software domain, operating systems interface with the hardware by handling details such as managing the window system, the file manager, and communicating with external devices like a printer or keyboard. Your computer probably runs on one of the operating systems macOS, Windows, or Linux and your phone on iOS or Android. The architecture of a computer offers a way for the hardware and software to communicate through a machine language, a set of low-level instructions that the hardware understands and that the software

translates from language that is relatively easily understood by humans to those instructions. Most computer processors today use an architecture called x86 which was popularized by IBM and Intel. Most phones use the low-power ARM architecture. The physical components of the computer (processor, memory) are built out of integrated circuits (also called chips) which are designed around logic gates. Logic gates are built out of transistors, capacitors, etc and operate on 0s and 1s. Over the next few lectures, we will focus on the lower levels of abstraction of computers and start building our way up.

Transistors

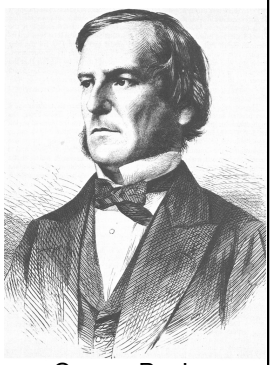
- Transistors are semiconductor devices made out of silicon.
- They act as switches that can be open or closed by applying electricity.
- Integrated circuits in modern computers are built off billions of transistors.
  - They are **small** – you can fit a few thousand of them across the width of a human hair!
  - They are **fast** – they can switch states millions of times per second.
  - They are **reliable** – they can run without any issue for decades.
- Invented in 1947 by Bardeen, Brattain, and Shockley at Bell Lab.
  - They shared the 1956 Nobel Prize in Physics for their achievement.
- A lot of development of transistors and circuits took place in Santa Clara Valley, which by the 1980s started being referred to as the **Silicon Valley**.



At the lowest level of abstraction, we have physical components that are used to make up modern computers. One such type is the transistor. Transistors are semiconductor devices made out of silicon that act as switches that can be open or closed by applying electrical power. Modern computers comprise of integrated circuits made out of billions of transistors! Transistors have a lot of advantages: they are small (as small as 2-3 nanometers!), extremely fast (they can switch between states rapidly), and reliable (they can keep running for decades). Transistors were invented in 1947 by Bardeen, Brattain, and Shockley (infamous for his eugenicist views) at the famous Bell Lab. The three of them shared a Nobel Prize in Physics in 1956 for their invention. Fun fact: Bardeen is the only person to have been awarded the Nobel Prize in Physics twice; the second time was in 1972 with Cooper and Schrieffer for their work on superconductivity. Because a lot of the development of transistors and circuits in general took place in the Santa Clara Valley, California, and because these computer parts are made out of silicon, the region took the name we all know today… Silicon Valley.

Binary representation and Boolean algebra

- Transistors are powered with electricity and can be in one of two states, allowing us to represent information in **binary**.
- Binary means two states. Common interchangeable representations are:
  - 0 and 1,
  - FALSE and TRUE,
  - LOW and HIGH.
- Working only with two states has the advantage of giving us a distinct signal, thus minimizing errors.
- It also works with a well-established branch of Mathematics called **Boolean algebra** that works with binary variables and logical operations.
  - George Boole was a self-taught English mathematician. He introduced **binary variables** and the logic operations **NOT, AND**, and **OR.**

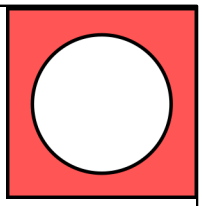


George Boole

4

We said that transistors in computers can have one of two states (on or off) which allows us to represent information in binary. Binary means two states. The off state (no electricity flow) is also commonly and interchangeable represented as 0, FALSE, or LOW. The on state (electricity flows) is equivalently represented as 1, TRUE, or HIGH. Working only with two states minimizes errors as it gives us a strong signal. It also allows us to work with a well-established branch of Mathematics developed by George Boole, called Boolean algebra. George Boole (who was self-taught) published in 1854 the work *Investigation of the Laws of Thought* which sought to systematically and formally prove logic statements that were true, going beyond Aristotle's philosophical approach to logic. In regular Algebra, variables are numbers and operators perform addition, multiplication, etc. In Boolean algebra, variables are binary and have one of two values, TRUE or FALSE which can be combined using logical operations, such as NOT, AND, and OR that we will cover next.

NOT operation – Negation



- Unary operator, i.e., it is applied to only one operand. Also denoted as ¬.
- It flips FALSE (0) to TRUE (1) and, equivalently, TRUE (1) to FALSE (0).
- We can use a **truth table** to enumerate all possible values of operands and the effect of the logical operations when applied to them.
- Truth table for NOT:

X	NOT X
FALSE	TRUE
TRUE	FALSE

- Equivalently:

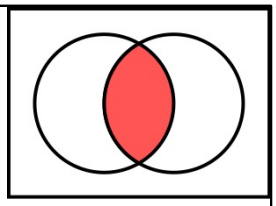
X	¬ X
0	1
1	0

- Logic gate (inverter) NOT: 

5

Let's start with the simplest operation that we can apply when working with binary information, that of NOT. NOT also known as Boolean negation is a unary operator, that is it is applied to only one operand. You can also see it represented ¬. What NOT does is that it flips FALSE to TRUE and TRUE to FALSE. Remember, TRUE/FALSE is equivalent to 1/0 so you can think of NOT flipping 0 to 1 and 1 to 0, too. You can see the *truth table* where it flips the state of X to its counterpart. Either truth table is correct depending on how you want to represent the binary values. In circuit design, we apply Boolean algebra to electronic hardware consisting of digital logic gates. The logic gate that represents NOT is called an inverter and is symbolized with a triangle (which means it passes the value it was given) and a circle next to it (which represents the inversion) . The line on the left corresponds to the input wire and the line on the right the output wire. Under the hood, the inverter (and all other logic gates), comprise of physical components like transistors, but we are already moving up the abstraction ladder and won't be seeing those details.

AND operation – Conjunction



- Binary operator, i.e., it is applied two operands. Also denoted as ∧.
- Produces TRUE (1) only if *both* operands are TRUE (1).
- Truth table for AND:

X	Y	X AND Y
FALSE	FALSE	FALSE
FALSE	TRUE	FALSE
TRUE	FALSE	FALSE
TRUE	TRUE	TRUE

- Equivalently:

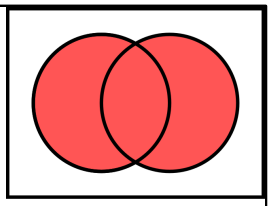
X	Y	X ∧ Y
0	0	0
0	1	0
1	0	0
1	1	1

- Logic gate AND: 

6

AND, also known as conjunction, is a binary operator, that is it is applied on two operands. It is also denoted as ∧. AND produces TRUE (1) only if both operands it is applied to are TRUE (1). Similarly to NOT, you can encounter two commonly used truth tables for AND. Note how only the last row where both X and Y are TRUE/1 produces TRUE/1. In circuit design, the logic gate that represents AND is shown above. Note that it receives two input wires that it applies AND on their electrical input and produces one output.

OR operation – Disjunction



- Binary operator, i.e., it is applied two operands. Also denoted as ∨.
- Produces TRUE (1) if *at least* one operand is TRUE (1).
- Truth table for OR:

X	Y	X OR Y
FALSE	FALSE	FALSE
FALSE	TRUE	TRUE
TRUE	FALSE	TRUE
TRUE	TRUE	TRUE

- Equivalently:

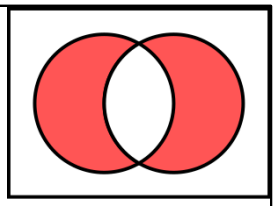
X	Y	X ∨ Y
0	0	0
0	1	1
1	0	1
1	1	1

- Logic gate OR: 

7

OR also known as disjunction is a binary operator, that is it is applied on two operands. It is also denoted as ∨. OR produces TRUE (1) if at least one of the operands it is applied to is TRUE (1). Similarly to NOT and AND, you can encounter two commonly used truth tables for OR. Note how only the first row where both X and Y are FALSE/0 produces FALSE/0, all other rows produce TRUE/1. In circuit design, the logic gate that represents OR is shown above. Note that similar to AND, it receives two wires that it applies OR on their electrical input and produces one output.

XOR operation – Exclusive OR



- Binary operator, i.e., it is applied two operands. Also denoted as $\oplus$.
- Produces TRUE (1) if *exactly* one operand is TRUE (1).
- Truth table for XOR:

X	Y	X XOR Y
FALSE	FALSE	FALSE
FALSE	TRUE	TRUE
TRUE	FALSE	TRUE
TRUE	TRUE	FALSE

- Equivalently:

X	Y	X $\oplus$ Y
0	0	0
0	1	1
1	0	1
1	1	0

- Logic gate XOR: 

8

Beyond these three basic operators that were introduced by Boole, there have been secondary ones that have been subsequently introduced, with XOR being the most popular. XOR, also known as exclusive OR, is a binary operator, that is it is applied on two operands. It is also denoted as $\oplus$. XOR produces TRUE (1) if exactly one of the operands it is applied to is TRUE (1). Similarly to NOT, AND, OR, you can encounter two commonly used truth tables for XOR. Note how only the second and third row where only one of X and Y is TRUE/1 produces TRUE/1, all other rows produce FALSE/0. In circuit design, the logic gate that represents XOR is shown above. Note, that similar to AND and OR, it receives two wires that it applies XOR on their electrical input and produces one output. It looks a bit like OR but it has that extra smile :) before the gate.

PRACTICE TIME – De Morgan's laws

- ¬ (X ∧ Y) = ¬ X ∨ ¬ Y
- ¬ (X ∨ Y) = ¬ X ∧ ¬ Y

X	Y	X ∧ Y	¬ (X ∧ Y)	X ∨ Y	¬ (X ∨ Y)	¬ X	¬ Y	¬ X ∧ ¬ Y	¬ X ∨ ¬ Y
0	0								
0	1								
1	0								
1	1								

9

We can use the truth tables we learned about the different logical operations to evaluate more complex logical expressions. Two such expressions are known as De Morgan's Law and provide handy equations. The first law states that the negation of X AND Y is the same as the NOT X OR NOT Y. The second law says that NOT (X OR Y) is the same as (NOT X) AND (NOT Y). I have provided you with the four different combinations that X and Y can take. Please fill the rest of the columns, using our understanding of how NOT, AND, and OR work.

ANSWER – De Morgan's laws

- ¬ (X ∧ Y) = ¬ X ∨ ¬ Y
- ¬ (X ∨ Y) = ¬ X ∧ ¬ Y

X	Y	X ∧ Y	¬ (X ∧ Y)	X ∨ Y	¬ (X ∨ Y)	¬ X	¬ Y	¬ X ∧ ¬ Y	¬ X ∨ ¬ Y
0	0	0	1	0	1	1	1	1	1
0	1	0	1	1	0	1	0	0	1
1	0	0	1	1	0	0	1	0	1
1	1	1	0	1	0	0	0	0	0

10

As you can see in the green highlighted columns, indeed NOT (X AND Y) matches (NOT X) OR (NOT Y). And in the orange highlighted columns, NOT(X OR Y) matches (NOT X) AND (NOT Y)

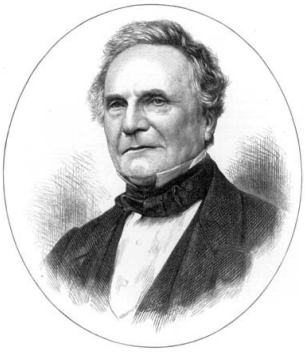
A mathematical lineage to the first computer



Augustus De Morgan



Ada Lovelace



Charles Babbage

11

Interesting tidbit of history. De Morgan's Laws are named after Augustus De Morgan (1806–1871), a British mathematician and logician who was also the math tutor of Ada Lovelace (1815–1852). As we saw last week, Ada Lovelace was an English mathematician and the only legitimate child of poet Lord Byron! She is known as the first computer programmer and she maintained correspondence and translated the work of English mathematician and inventor Charles Babbage (1791-1871) who worked on the idea of what is considered the first digital computer, the Analytical Engine, which would be a general-purpose, fully program-controlled computer.

PRACTICE TIME – Evaluate these expressions

- $0 \vee (\neg 0)$
- $1 \wedge (\neg 1)$
- $\neg (1 \wedge 0)$
- $0 \wedge (\neg 1)$
- $\neg (1 \vee (\neg 1))$
- $1 \oplus (\neg 1)$

12

A bit more practice time. Using our understanding of how NOT, AND, OR, and XOR work, write what the results of evaluating the following expressions would be.

ANSWER– Evaluate these expressions

- $0 \vee (\neg 0) = 0 \vee 1 = 1$
- $1 \wedge (\neg 1) = 1 \wedge 0 = 0$
- $\neg (1 \wedge 0) = \neg 0 = 1$
- $0 \wedge (\neg 1) = 0 \wedge 0 = 0$
- $\neg (1 \vee (\neg 1)) = \neg (1 \vee 0) = \neg 1 = 0$
- $1 \oplus (\neg 1) = 1 \oplus 0 = 1$

13

Boolean functions of three+ variables

- AND($X_1, X_2, \ldots, X_n$) = 0 if any argument is 0, 1 if all arguments are 1.
- OR($X_1, X_2, \ldots, X_n$) = 1 if any argument is 1, 0 if all arguments are 0.
- Infinite possibilities. Here are two examples of such Boolean functions:
  - MAJ($X_1, X_2, \ldots, X_n$) = 1 if (strictly) more arguments are 1 than 0, 0 otherwise.
  - ODD($X_1, X_2, \ldots, X_n$) = 1 if an odd number of arguments is 1, 0 otherwise.

X	Y	Z	AND(X,Y,Z)	OR(X,Y,Z)	MAJ(X,Y,Z)	ODD(X,Y,Z)
0	0	0	0	0	0	0
0	0	1	0	1	0	1
0	1	0	0	1	0	1
0	1	1	0	1	1	0
1	0	0	0	1	0	1
1	0	1	0	1	1	0
1	1	0	0	1	1	0
1	1	1	1	1	1	1

14

So far we have seen that you can apply AND and OR on two arguments. In practice, you can write Boolean functions with more than two variables. For example, AND($X_1, X_2, \ldots, X_n$) would evaluate to 0 if any argument is 0. It would only evaluate to 1 if all arguments were 1. OR($X_1, X_2, \ldots, X_n$) evaluates to 1 as long as least one argument is 1. If all arguments are 0, then it evaluates to 0. There are infinite possibilities of what we can do with Boolean functions. Here are two more functions. MAJ($X_1, X_2, \ldots, X_n$) evaluates to 1 if strictly more arguments are 1 than 0, while ODD($X_1, X_2, \ldots, X_n$) evaluates to 1 if an odd number of arguments is 1. You can see the truth tables for all those Boolean functions for three variables here.

Disjunctive and conjunctive normal form

- **Theorem (Boole, 1847):** Any Boolean function can be represented as a disjunction (one OR) of conjunctions (a bunch of ANDs) of its arguments and their negation (NOT). This is also known as **disjunctive normal form** (DNF) or a sum of products/minterms.
- Equivalently, any Boolean function can be represented as a conjunction (one AND) of disjunctions (a bunch of ORs) of its arguments and their negation (NOT). This is also known as a **conjunctive normal form** (CNF) or product of sums/maxterms.
- How to find the disjunctive normal form of a function:
  - Using truth tables, we isolate the rows where the function evaluates to TRUE. Each row contributes a conjunction of the arguments (if TRUE) or their negation (if FALSE).
- How to find the conjunctive normal form of a function:
  - Using truth tables, we isolate the rows where the function evaluates to FALSE. Each row contributes a disjunction of the arguments (if FALSE) or their negation (if TRUE).

15

Along with his Algebra, Boole wrote a very interesting theorem in 1847. Any Boolean function can be represented as a disjunction of conjunctions (one OR of a bunch of ANDs) of its arguments and their negation (NOT). This is also known as **disjunctive normal form** (DNF) or a sum of products/minterms. This cab be equivalently seen, as any Boolean function can be represented as a conjunction of disjunctions (one AND of a bunch of ORs) of its arguments and their negation (NOT). This is also known as a **conjunctive normal form** (CNF) or product of sums/maxterms.

To find the disjunctive normal form of a function, we use the truth tables, isolate the rows where the function evaluates to TRUE. Each row contributes to the OR an AND statement of the arguments (if TRUE) or their negation (if FALSE).

To find the conjunctive normal form of a function, we use the truth tables, isolate the rows where the function evaluates to FALSE. Each row contributes to the AND an OR statement of the arguments (if FALSE) or their negation (if TRUE).

Minterms

Minterm: a Boolean expression consisting of the conjunction (AND) of all variables of a Boolean function. Each variable will appear exactly once in either its true (if the variable was 1) or complemented form (if the variable was 0). The minterm is formed using the values of the variables which make the value of the minterm equal to 1.

Row number	X	Y	Z	Minterm
0	0	0	0	$m_0 = \neg X \wedge \neg Y \wedge \neg Z$
1	0	0	1	$m_1 = \neg X \wedge \neg Y \wedge Z$
2	0	1	0	$m_2 = \neg X \wedge Y \wedge \neg Z$
3	0	1	1	$m_3 = \neg X \wedge Y \wedge Z$
4	1	0	0	$m_4 = X \wedge \neg Y \wedge \neg Z$
5	1	0	1	$m_5 = X \wedge \neg Y \wedge Z$
6	1	1	0	$m_6 = X \wedge Y \wedge \neg Z$
7	1	1	1	$m_7 = X \wedge Y \wedge Z$

16

Before we see how to construct the disjunctive and conjunctive normal forms of a Boolean function, we will talk about how we can find its minterms and maxterms. A minterm is a Boolean logical expression consisting of the conjunction (AND) of all variables, each appearing exactly once in either its true (if the variable was 1) or complemented form (if the variable was 0). The minterm is formed using the values of the variables which make the value of the minterm equal to 1. We symbolize minterms with small m subscripted from 0 to to (2^n)-1, where n is the number of the variables of the Boolean function. For example, here, m_0 corresponds to the 0^{th} row where all three variables are false, as such the minterm is going to be not X and not Y and not Z.

Maxterms

Maxterm: a Boolean expression consisting of the disjunction (OR) of all variables of a Boolean function. Each variable will appear exactly once in either its true (if the variable was 0) or complemented form (if the variable was 1). The maxterm is formed using the values of the variables which make the value of the maxterm equal to 0.

Row number	X	Y	Z	Minterm	Maxterm
0	0	0	0	$m_0 = \neg X \wedge \neg Y \wedge \neg Z$	$M_0 = X \vee Y \vee Z$
1	0	0	1	$m_1 = \neg X \wedge \neg Y \wedge Z$	$M_1 = X \vee Y \vee \neg Z$
2	0	1	0	$m_2 = \neg X \wedge Y \wedge \neg Z$	$M_2 = X \vee \neg Y \vee Z$
3	0	1	1	$m_3 = \neg X \wedge Y \wedge Z$	$M_3 = X \vee \neg Y \vee \neg Z$
4	1	0	0	$m_4 = X \wedge \neg Y \wedge \neg Z$	$M_4 = \neg X \vee Y \vee Z$
5	1	0	1	$m_5 = X \wedge \neg Y \wedge Z$	$M_5 = \neg X \vee Y \vee \neg Z$
6	1	1	0	$m_6 = X \wedge Y \wedge \neg Z$	$M_6 = \neg X \vee \neg Y \vee Z$
7	1	1	1	$m_7 = X \wedge Y \wedge Z$	$M_7 = \neg X \vee \neg Y \vee \neg Z$

17

Conversely, a maxterm is a Boolean expression consisting of the disjunction (OR) of all variables of a Boolean function. Each variable will appear exactly once in either its true (if the variable was 0) or complemented form (if the variable was 1). The maxterm is formed using the values of the variables which make the value of the maxterm equal to 0. We symbolize maxterms with capital M subscripted from 0 to to (2^n)-1, where n is the number of the variables of the Boolean function. For example, here, M_0 corresponds to the 0^{th} row where all three variables are false, as such the maxterm is going to be X or Y or Z.

Example – DNF of MAJ function

- Remember, MAJ($X_1, X_2, \ldots, X_n$) = 1 if (strictly) more arguments are 1 than 0, 0 otherwise.

X	Y	Z	Minterm	Maxterm	MAJ(X,Y,Z)
0	0	0	$m_0 = \neg X \wedge \neg Y \wedge \neg Z$	$M_0 = X \vee Y \vee Z$	0
0	0	1	$m_1 = \neg X \wedge \neg Y \wedge Z$	$M_1 = X \vee Y \vee \neg Z$	0
0	1	0	$m_2 = \neg X \wedge Y \wedge \neg Z$	$M_2 = X \vee \neg Y \vee Z$	0
0	**1**	**1**	$\mathbf{m_3 = \neg X \wedge Y \wedge Z}$	$M_3 = X \vee \neg Y \vee \neg Z$	**1**
1	0	0	$m_4 = X \wedge \neg Y \wedge \neg Z$	$M_4 = \neg X \vee Y \vee Z$	0
1	**0**	**1**	$\mathbf{m_5 = X \wedge \neg Y \wedge Z}$	$M_5 = \neg X \vee Y \vee \neg Z$	**1**
1	**1**	**0**	$\mathbf{m_6 = X \wedge Y \wedge \neg Z}$	$M_6 = \neg X \vee \neg Y \vee Z$	**1**
1	**1**	**1**	$\mathbf{m_7 = X \wedge Y \wedge Z}$	$M_7 = \neg X \vee \neg Y \vee \neg Z$	**1**

- To derive the disjunctive normal form of a function, we will isolate the rows where the output is equal to 1. We will then connect with a disjunction (OR) the corresponding minterms.
- For the MAJ Boolean function over three variables, this will result in:
  DNF of MAJ(X, Y, Z) = $m_3 \vee m_5 \vee m_6 \vee m_7 =$ (¬X ∧ Y ∧ Z) ∨ (X ∧ ¬Y ∧ Z) ∨ (X ∧ Y ∧ ¬Z) ∨ (X ∧ Y ∧ Z)

18

Now that we calculated the minterms and maxterms we can use them to calculate the disjunctive normal form (DNF) and conjunctive normal form (CNF) that describe this Boolean function. In general, to derive the disjunctive normal form of a function, we will isolate the rows where the output is equal to 1. We will then connect with a disjunction (OR) the corresponding minterms. For the MAJ Boolean function (returns 1 if strictly more arguments are 1 other than 0, 0 otherwise) over three variables, this will result in:
DNF of MAJ(X, Y, Z) = $m_3 \vee m_5 \vee m_6 \vee m_7 =$ (¬X ∧ Y ∧ Z) ∨ (X ∧ ¬Y ∧ Z) ∨ (X ∧ Y ∧ ¬Z) ∨ (X ∧ Y ∧ Z)

Example – CNF of MAJ function

- To derive the conjunctive normal form of a function, we will isolate the rows where the output is equal to 0. We will then connect with a conjunction (AND) the corresponding maxterms.

X	Y	Z	Minterm	Maxterm	MAJ(X,Y,Z)
0	0	0	$m_0 = \neg X \wedge \neg Y \wedge \neg Z$	$\boldsymbol{M_0 = X \vee Y \vee Z}$	**0**
0	0	1	$m_1 = \neg X \wedge \neg Y \wedge Z$	$\boldsymbol{M_1 = X \vee Y \vee \neg Z}$	**0**
0	1	0	$m_2 = \neg X \wedge Y \wedge \neg Z$	$\boldsymbol{M_2 = X \vee \neg Y \vee Z}$	**0**
0	1	1	$m_3 = \neg X \wedge Y \wedge Z$	$M_3 = X \vee \neg Y \vee \neg Z$	1
1	0	0	$m_4 = X \wedge \neg Y \wedge \neg Z$	$\boldsymbol{M_4 = \neg X \vee Y \vee Z}$	**0**
1	0	1	$m_5 = X \wedge \neg Y \wedge Z$	$M_5 = \neg X \vee Y \vee \neg Z$	1
1	1	0	$m_6 = X \wedge Y \wedge \neg Z$	$M_6 = \neg X \vee \neg Y \vee Z$	1
1	1	1	$m_7 = X \wedge Y \wedge Z$	$M_7 = \neg X \vee \neg Y \vee \neg Z$	1

- For the MAJ Boolean function over three variables, this will result in:
  CNF of MAJ(X, Y, Z) = $M_0 \wedge M_1 \wedge M_2 \wedge M_4$ = (X ∨ Y ∨ Z) ∧ (X ∨ Y ∨ ¬Z) ∧ (X ∨ ¬Y ∨ Z) ∧ (¬X ∨ Y ∨ Z)

19

Conversely, to derive the conjunctibe normal form of a function, we will isolate the rows where the output is equal to 0. We will then connect with a conjunction (AND) the corresponding maxterms. For the MAJ Boolean, this will result in:
CNF of MAJ(X, Y, Z) = $M_0 \wedge M_1 \wedge M_2 \wedge M_4$ = (X ∨ Y ∨ Z) ∧ (X ∨ Y ∨ ¬Z) ∧ (X ∨ ¬Y ∨ Z) ∧ (¬X ∨ Y ∨ Z)

PRACTICE TIME - ODD function

- Remember, ODD($X_1, X_2, \ldots, X_n$) = 1 if an odd number of arguments is 1, 0 otherwise.

X	Y	Z	Minterm	Maxterm	ODD(X,Y,Z)
0	0	0	$m_0 =$	$M_0 =$	0
0	0	1	$m_1 =$	$M_1 =$	1
0	1	0	$m_2 =$	$M_2 =$	1
0	1	1	$m_3 =$	$M_3 =$	0
1	0	0	$m_4 =$	M_4	1
1	0	1	$m_5 =$	$M_5 =$	0
1	1	0	$m_6 =$	$M_6 =$	0
1	1	1	$m_7 =$	$M_7 =$	1

- Disjunctive normal form (OR of minterms for rows that are 1) :
  ODD(X, Y, Z) =
- Conjunctive normal form (AND of maxterms for rows that are 0):
  ODD(X, Y, Z) =

20

Now is your turn. Applying the same process, find what the DNF and CNF of the ODD function is.

ANSWER - ODD function

- Remember, ODD($X_1, X_2, \ldots, X_n$) = 1 if an odd number of arguments is 1, 0 otherwise.

X	Y	Z	Minterm	Maxterm	ODD(X,Y,Z)
0	0	0	$m_0 = \neg X \wedge \neg Y \wedge \neg Z$	$M_0 = X \vee Y \vee Z$	0
0	0	1	$m_1 = \neg X \wedge \neg Y \wedge Z$	$M_1 = X \vee Y \vee \neg Z$	1
0	1	0	$m_2 = \neg X \wedge Y \wedge \neg Z$	$M_2 = X \vee \neg Y \vee Z$	1
0	1	1	$m_3 = \neg X \wedge Y \wedge Z$	$M_3 = X \vee \neg Y \vee \neg Z$	0
1	0	0	$m_4 = X \wedge \neg Y \wedge \neg Z$	$M_4 = \neg X \vee Y \vee Z$	1
1	0	1	$m_5 = X \wedge \neg Y \wedge Z$	$M_5 = \neg X \vee Y \vee \neg Z$	0
1	1	0	$m_6 = X \wedge Y \wedge \neg Z$	$M_6 = \neg X \vee \neg Y \vee Z$	0
1	1	1	$m_7 = X \wedge Y \wedge Z$	$M_7 = \neg X \vee \neg Y \vee \neg Z$	1

- Disjunctive normal form (OR of minterms for rows that are 1) :
  ODD(X, Y, Z) = $m_1 \vee m_2 \vee m_4 \vee m_7$ = (¬X ∧ ¬Y ∧ Z) ∨ (¬X ∧ Y ∧ ¬Z) ∨ (X ∧ ¬Y ∧ ¬Z) ∨ (X ∧ Y ∧ Z)
- Conjunctive normal form (AND of maxterms for rows that are 0):
  ODD(X, Y, Z) = $M_0 \wedge M_3 \wedge M_5 \wedge M_6$ = (X ∨ Y ∨ Z) ∧ (X ∨ ¬Y ∨ ¬Z) ∧ (¬X ∨ Y ∨ ¬Z) ∧ (¬X ∨ ¬Y ∨ Z)

21

You should have gotten the following.

Boolean variables in Python

- In Python, Boolean variables are represented through the `bool` data type. A bool variable can either be `True` or `False`. These values can also be seen as equal to 1 and 0, respectively.
- For example, `raining = False` declares a variable called `raining` and assigns to it the value `False`.
- The comparison operators ==, !=, >, <, >=, <= return bools. For example,
  - `10 < 0` evaluates to `False`
  - `11 >= 11` evaluates to `True`
  - `11 > 11.0` evaluates to `False`
  - `10 == 10.0` evaluates to `True`
  - `10 != 10.0` evaluates to `False`

22

So far we have been abstractly talking about 0s and 1s, but in Python, you have already worked with Boolean variables through the bool data type. A bool variable can be True or False, equivalently to 1 and 0. If you have worked with comparison operators, you have also evaluated Boolean expressions.

Boolean Algebra in Python

- We can combine bools and logical operators in Boolean expressions, for example, to evaluate them in **conditionals** (`if` statements) and **loops** (`for` and `while`).
- Python supports three logical operators:
  - `not` for negation
  - `and` for conjunction
  - `or` for disjunction

23

We can combine bool variables and logical operators in Boolean expressions, for example to evaluate them in conditionals (if statements) and loops, such as the for and while loops. Python also supports not, and, and or.

Short-circuit evaluation in Python

- Python supports **short-circuiting**, a technique that evaluates a logical expression from left to right and stops once the truth can be determined.
  - When an expression with an `or` statement gets evaluated, it is enough to just see the first operand evaluate to `True` and return it. No need to check the second operand because `True or x` will evaluate to `True` no matter what `x` evaluates to.
  - When an expression with an `and` statement gets evaluated, it is enough to just see the first operand evaluate to `False` and return it. No need to check the second operand because `False and x`, will evaluate to `False` no matter what `x` evaluates to.

24

Interestingly, Python supports short-circuiting, a technique that evaluates a logical expression from left to right and stops once the truth can be determined without wasting unnecessarily time. For example, if you have an or statement and you find a true operand, then you can stop because you know that only one needs to be TRUE. Similarly, if you evaluate an expression with an and statement and the first operand is False, you can immediately answer that the result is False.

Short-circuit evaluation and optimization

- Because of short-circuit evaluation, we can avoid deeply nested if statements to handle exceptional cases. For example,
  - `if x == 0 or (x-1)/x > 0.5: …`
  - When x is equal to 0, evaluating the second operand of the or statement would cause a divide-by-zero error. But the second disjunct isn't evaluated when x is equal to 0 because the first operand was True!
- We can take advantage of short-circuit evaluation to make our code a bit faster. For example, when writing an if statement that contains the logical operator *and*, you should first write what is simpler to calculate or often false before something that is complex to calculate or often true:
  - `if (simpleOrOftenFalse() and complexOrOftenTrue()):…`

25

Short-circuiting allows us to write more condensed if statements or even make our code faster.

PRACTICE TIME – Boolean expressions

- Assume the variable x currently stores the value 47. What will the following expressions evaluate to?
  - `True and not False`
  - `not True or not False`
  - `True and not 0`
  - `False or not 1`
  - `2 < x and x < 50`
  - `x > 2 and < 20`
  - `(2 > x) or (x == 47)`
  - `not (x == x)`
  - `not (x != 3)`
  - `x < 0 and (y/x) == 3`
  - `x > 0 or (y/x) == 3`

26

Let's see whether you can evaluate the following Boolean expressions

ANSWER – Boolean expressions

- Assume the variable x currently stores the value 47. What will the following expressions evaluate to?
  - `True and not False - True`
  - `not True or not False - True`
  - `True and not 0 – True`
  - `False or not 1 - False`
  - `2 < x and x < 50 - True`
  - `x > 2 and < 20 – Invalid syntax`
  - `(2 > x) or (x == 47) - True`
  - `not (x == x) - False`
  - `not (x != 3) – False`
  - `x < 0 and (y/x) == 3 – False`
  - `x > 0 or (y/x) == 3 - True`

27

PRACTICE TIME – Conditionals

- Rewrite the following code to make it cleaner and avoid the use of nested if statements:

```
def get_discount(member, purchase_amount):
	if member == True:
		if purchase_amount > 100:
			return "Discount applied"
		else:
			return "No discount"
	else:
		if purchase_amount > 200:
			return "Discount applied"
		else:
			return "No discount"

print(get_discount(True, 120)) # Expected: "Discount applied"
print(get_discount(True, 80)) # Expected: "No discount"
print(get_discount(False, 220)) # Expected: "Discount applied"
print(get_discount(False, 50)) # Expected: "No discount"
```

28

Can you write the following function in a more condensed way? I have provided you the following print statements and what you should expect them to produce

ANSWER – Conditionals

- Rewrite the following code to make it cleaner and avoid the use of nested if statements:

```
def get_discount(member, purchase_amount):
	if (purchase_amount > 200) or (member and purchase_amount > 100):
		return "Discount applied"
	else:
		return "No discount"

print(get_discount(True, 120)) # Expected: "Discount applied"
print(get_discount(True, 80)) # Expected: "No discount"
print(get_discount(False, 220)) # Expected: "Discount applied"
print(get_discount(False, 50)) # Expected: "No discount"
```

29

Here is a condensed version that accomplishes the same thing

Practice Problems – Problem 1

- You are given the following truth table for a Boolean function with three variables, A, B, and C, and an output column F.
- What are the disjunctive and conjunctive normal forms based on this truth table?

A	B	C	F
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

30

Practice Problems – Problem 2

Assuming x = 6, y = 2, z = 12. What would the following Python expressions evaluate to?

- x == 10 or x < 5
- x != 7 and x > 3
- not (x >= 5 and y <= 2)
- (x < 0) or (x >= 0)
- not (x != x)
- x > 1 and (z / x) != 2
- x <= 0 or (z / x) == 1
- x == 4 and y == 4

Practice Problems – Answer 1

- Disjunctive normal form:
  $F = (\neg A \wedge \neg B \wedge \neg C) \vee (\neg A \wedge B \wedge \neg C) \vee (A \wedge \neg B \wedge C) \vee (A \wedge B \wedge \neg C)$
- Conjunctive normal form:
  $F = (A \vee B \vee \neg C) \wedge (A \vee \neg B \vee \neg C) \wedge (\neg A \vee B \vee C) \wedge (\neg A \vee \neg B \vee \neg C)$

A	B	C	F
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

32

Practice Problems – Answer 2

Assuming x = 6, y = 2, z = 12. What would the following Python expressions evaluate to?

- x == 10 or x < 5 → 6 == 10 or 6 < 5 → False or False → **False**
- x != 7 and x > 3 → 6 != 7 and 6 > 3 → True and True → **True**
- not (x >= 5 and y <= 2) → not (6 >= 5 and 2 <= 2) → not (True and True) → not True → **False**
- (x < 0) or (x >= 0) → 6 < 0 or 6 >= 0 → False or True → **True** (always true, no matter what x is)
- not (x != x) → not False → **True**
- x > 1 and (z / x) != 2 → 6 > 1 and (12 / 6 != 2) → True and False → **False**
- x <= 0 or (z / x) == 1 → 6 <= 0 or (12 / 6 == 1) → False or False → **False**
- x == 4 and y == 4 → 6 == 4 and 2 == 4 → False and False → **False**