Lesson 3: Storing and manipulating unsigned integers

- In Lessons 1&2 of this class, we explored many features of the MATLAB Desktop including using the Command Window as a basic Calculator. We also saw how to write programs using .m script files

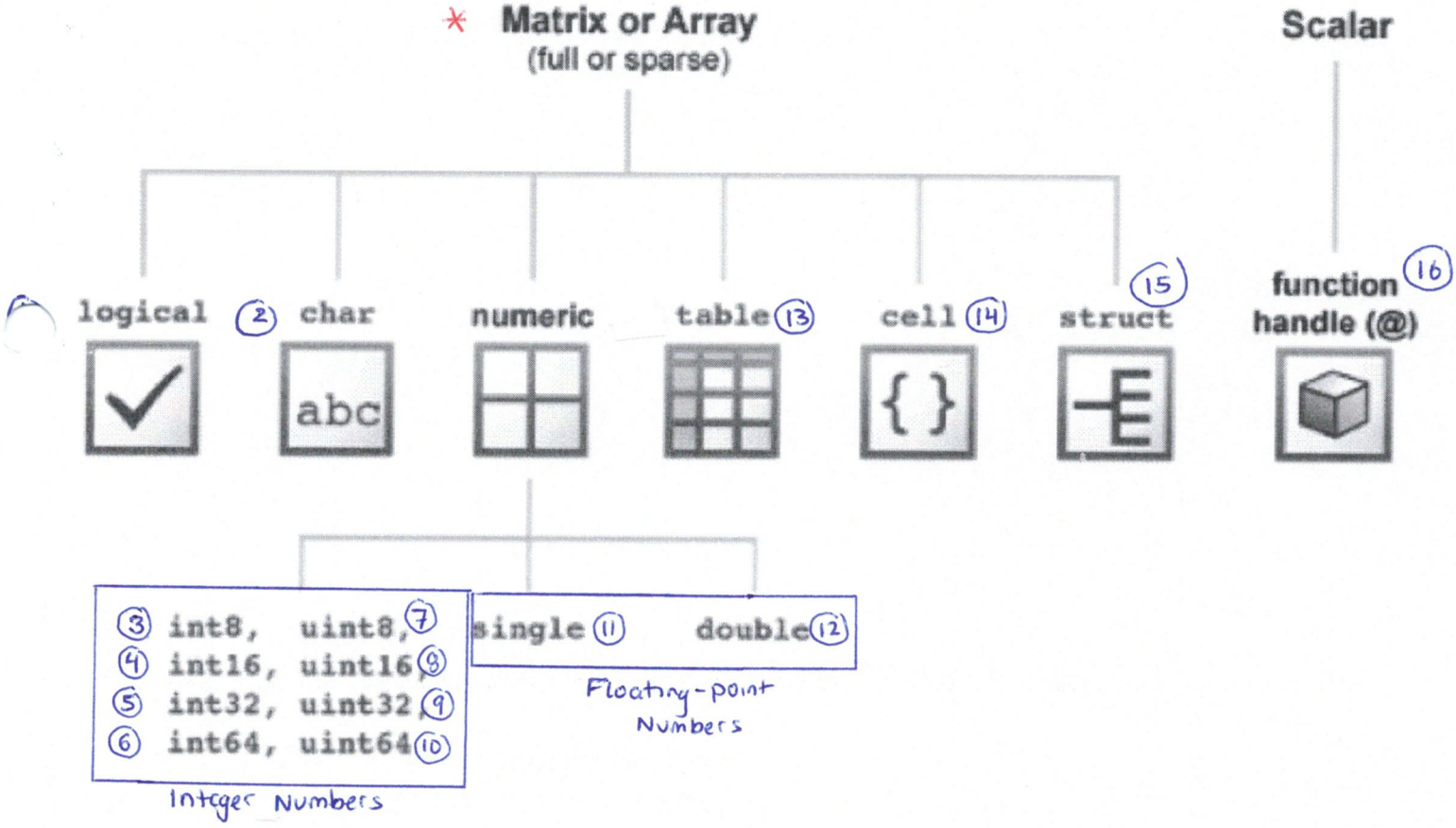
- In all of the work we did in Lesson 1, we focused on storing, manipulating, and analyzing scalars. In other words, all of the mathematical operations we performed and variables we saved were scalars (individual numbers)

- In Lesson 3 we expand our perspective and explore how we can use MATLAB to store unsigned integers.

lly speaking, an unsigned integer is a nonnegative whole number. As we will see, we need to develop familiarity with multiple number systems used to represent such numbers.

Before we dive into a deeper focus on number systems, let's consider a brief overview of the 16 different data types, known as classes, that we can use in MATLAB (seen in Fig 1 below). For more about this, see MATLAB's Documentation entry on "Fundamental MATLAB Classes".

All of the fundamental MATLAB classes are shown in the diagram below:



Each of these classes provides a specialized structure for storing data in MATLAB. The main idea behind providing multiple classes for data storage is that if we use consistent structures to store similar types of data, then we can design and execute specialized functions that take advantage of these structures.

* Note:

- Notice that 15 of the 16 data classes introduced on the last page are stored as arrays. This is a unique feature of MATLAB as compared with other programming languages. We will talk a lot more about this in future lessons.

- By default, MATLAB stores all numeric values in an array each of whose individual entries are stored as double-precision floating-point values (using 64 bits of memory and conforming to IEEE Standard 754 for double precision). Don't worry about all this technical jargon right now. We'll return to this later.

- However, depending on the context, we may want to store data as a different class. A brief overview of the 16 fundamental classes in MATLAB can be found below.

Class Name	Documentation	Intended Use
`double`, `single`	Floating-Point Numbers	• Required for fractional numeric data. • Double and Single precision. • Use `realmin` and `realmax` to show range of values. • Two-dimensional arrays can be sparse. • Default numeric type in MATLAB.
`int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `int64`, `uint64`	Integers	• Use for signed and unsigned whole numbers. • More efficient use of memory. • Use `intmin` and `intmax` to show range of values. • Choose from 4 sizes (8, 16, 32, and 64 bits).
`char`, `string`	Characters and Strings	• Data type for text. • Native or Unicode®. • Converts to/from numeric. • Use with regular expressions. • For multiple character arrays, use cell arrays. • Starting in R2016b, you also can store text in string arrays. For more information, see `string`.
`logical`	Logical Operations	• Use in relational conditions or to test state. • Can have one of two values: `true` or `false`. • Also useful in array indexing. • Two-dimensional arrays can be sparse.
`function_handle`	Function Handles	• Pointer to a function. • Enables passing a function to another function • Can also call functions outside usual scope. • Use to specify graphics callback functions. • Save to MAT-file and restore later.
`table`	Tables	• Rectangular container for mixed-type, column-oriented data. • Row and variable names identify contents. • Use the properties of a table to store metadata such as variable units. • Manipulation of elements similar to numeric or logical arrays. • Access data by numeric or named index. • Can select a subset of data and preserve the table container or can extract the data from a table.

Class Name	Documentation	Intended Use
`struct`	Structures	• Fields store arrays of varying classes and sizes. • Access one or all fields/indices in single operation. • Field names identify contents. • Method of passing function arguments. • Use in comma-separated lists. • More memory required for overhead
`cell`	Cell Arrays	• Cells store arrays of varying classes and sizes. • Allows freedom to package data as you want. • Manipulation of elements is similar to numeric or logical arrays. • Method of passing function arguments. • Use in comma-separated lists. • More memory required for overhead

- In this lesson (and in most of this class), we will focus heavily on storing and manipulating numerical data (mostly as floating-point numbers).

- However, the main points of this brief introduction to MATLAB data classes are:

  [1] Data classes provide pre-defined, specialized structure to allow us to store and manipulate different types of data more easily

  [2] When we create, store, and manipulate data in MATLAB, we should get into the habit of identifying and utilizing the class attributes of this data.

- Now that we have taken a brief look at the 16 fundamental MATLAB data classes, lets focus on storing numeric data in the form of unsigned integers.

- This lesson is our introduction to the numerous data classes that MATLAB provides for numeric data.

☐ Let's begin by discussing the most popular and general number systems in use today:

Name of Number System	Symbol and set notation	Description
Natural Numbers	$\mathbb{N} = \{1, 2, 3, \dots\}$	• the set of all positive (unsigned) whole numbers • used for counting (addition) and to order objects
Integers	$\mathbb{Z} = \{0, 1, -1, 2, -2, \dots\}$ $= \{\dots, -2, -1, 0, 1, 2, \dots\}$	• an extension of the natural numbers that includes all positive whole numbers, all negative whole numbers and the number zero • Used to extend the natural numbers to allow for both positive and negative (signed) integers and thus enable subtract
Rational numbers	$\mathbb{Q} = \{\frac{p}{q} : p, q \in \mathbb{Z} \text{ and } q \neq 0\}$ $= \{$fractions with integers in the numerator and denominator where denom cannot be zero$\}$	• an extension of the integers to include all fractions in the form $\frac{p}{q}$ ← numerator, ← denominator where both the numerator and denominator are integers $p, q \in \mathbb{Z}$ with $q \neq 0$. • Used to extend the integers to enable multiplication and division

Name of number system	symbol and set notation	Description
Real numbers	$\mathbb{R} = \{x : x \text{ is a real number}\}$ building a set theoretic definition of the real number system is not easy. In fact, this is the subject of a year-long, upper division course called Real analysis	• an extension of the rational numbers to include irrational (and transcendental) numbers. • used to extend the rational numbers to enable nth roots and the solution to a wide range of algebraic, trigonometric, geometric, calculus-based and differential problems
Complex numbers	$\mathbb{C} = \{a + b \cdot i : a, b \in \mathbb{R} \text{ and } i = \sqrt{-1}\}$	• an extension of the real numbers to include so-called imaginary numbers • Used to extend the real numbers to provide solutions to a larger class of algebraic equations i.e. $x^2 = -1$ and to provide a larger tool set for dynamical systems.

Nonnegative Numbers and the decimal System

- Let's take a deeper look at the set of nonnegative integers and the way we use numerical symbols to represent the nonnegative integers $\{0, 1, 2, 3, \ldots\}$

- In our entire mathematical career, we've spent lots of time learning to represent numbers using a system based on decimal* digits (the so-called decimal numeral system)

- In the decimal numeral system, we can represent any nonnegative integer as a string of digits chosen from the set of decimal digits

$$D = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

*Note:

- As we will see, the word decimal is synonomous with the phrase "base 10".

Example 1: Let's begin by looking at a two-digit nonnegative (unsigned) decimal integer

$$y = \underset{\text{digit 2}}{1}\ \underset{\text{digit 1}}{5}$$

How to count digits?

- when counting the number of digits, we start with the right-most digit, calling this digit the "first" digit of our number and increment our count moving left. In this case, we have a two-digit number.

Notice, that each "digit" is a nonnegative integer chosen from the set $D = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ of decimal digits and that the "total" value of the number y depends on both the numeric value of each digit combined with the "position" of each digit within the number.

- The origin of the word digit is from the latin word digitus meaning fingers or toe. This terminology arose from the practice of counting on the fingers.

Example 1: Let's take a look at a two-digit nonnegative (unsigned) decimal integer

$$y = \underset{\text{digit 2} \to \text{position 1}}{1}\ \underset{\text{digit 1} \to \text{position 0}}{5}$$

$$= 10 + 5$$

$$= 1 \cdot 10 + 5 \cdot 1$$

$$= \underbrace{1 \cdot 10^1}_{\text{2nd digit is "positioned" next to factor of base 10 to the power of 1}} + \underbrace{5 \cdot 10^0}_{\text{1st digit is "positioned" next to factor of base 10 to the power of 0}}$$

$$= d_1 \cdot 10^1 + d_0 \cdot 10^0$$

where $d_1 = 1$ and $d_0 = 5$. This number means 1 tens and 5 ones.

How to assign the "position" of a digit?

- The "position" of a digit is the value of the power of base 10 associated with this digit in our decimal expansion:
  - digit 1 is multiplied by 10^0 so has a 0th power of base 10 (and is thus said to be in position 0)
  - digit 2 is multiplied by 10^1 so has a 1st power of base 10 and is thus said to be in position 1.

Example 2: Next, let's consider a three-digit nonnegative (unsigned) decimal integer

$y = 255$ (digit 3 → 2 ← position 2; digit 2 → 5 ← position 1; digit 1 → 5 ← position 0)

$= 200 + 50 + 5$

$= 2 \cdot 100 + 5 \cdot 10 + 5 \cdot 1$

$= \boxed{2}\ 10^{\boxed{2}} + \boxed{5} \cdot 10^{\boxed{1}} + \boxed{5} \cdot 10^{\boxed{0}}$

(position 2 ↓ 2; position 1 ↓ 1; position 0 ↓ 0; digit 3 ↑ 2; digit 2 ↑ 5; digit 1 ↑ 5)

$= d_2 \cdot 10^2 + d_1 \cdot 10^1 + d_0 \cdot 10^0$

where $d_2 = 2$, $d_1 = 5$, and $d_0 = 5$. This number means 2 hundreds, 5 tens, and 5 ones.

How to count positions?

When counting the positions of each digit in a nonnegative (unsigned) decimal integer, we start with the right-most digit, saying this digit is in the zeroth position, and increment our position number moving left.

Example 3: We continue our work by considering a five-digit nonnegative (unsigned) decimal integer

digit 5 →	digit 4 →	digit 3 →	digit 2 →	digit 1 →
6	5	5	3	5
↑ position 4	↑ position 3	↑ position 2	↑ position 1	↑ position 0

$$
\begin{array}{rrrrrr}
= & 6 & 0 & 0 & 0 & 0 \\
 & & 5 & 0 & 0 & 0 \\
 & & & 5 & 0 & 0 \\
 & & & & 3 & 0 \\
+ & & & & & 5 \\
\hline
\end{array}
$$

What is the relationship between the digit number and the position of that digit?

- Notice that when counting digits and assigning positions, the digit number does not exactly match the position number.
- However, there is a discernable pattern: the position number is one less than the digit number.
- The reason for this difference is that each digit represents a scalar factor of a power of 10 and the position number encodes which power value is associated with each digit

$$= 6 \cdot 10000 + 5 \cdot 1000 + 5 \cdot 100 + 3 \cdot 10 + 5 \cdot 1$$

$$= 6 \cdot 10^4 + 5 \cdot 10^3 + 5 \cdot 10^2 + 3 \cdot 10^1 + 5 \cdot 10^0$$

$$= d_4 \cdot 10^4 + d_3 \cdot 10^3 + d_2 \cdot 10^2 + d_1 \cdot 10^1 + d_0 \cdot 10^0$$

where $d_4 = 6$, $d_3 = 5$, $d_2 = 5$, $d_1 = 3$, and $d_0 = 5$.

Example 4: Finally let's consider a ten-digit, nonnegative (unsigned) decimal integer

position 9	position 8	position 7	position 6	position 5	position 4	position 3	position 2	position 1	position 0
4	2	9	4	9	6	7	2	9	5
= 4	0	0	0	0	0	0	0	0	0
	2	0	0	0	0	0	0	0	0
		9	0	0	0	0	0	0	0
			4	0	0	0	0	0	0
				9	0	0	0	0	0
					6	0	0	0	0
						7	0	0	0
							2	0	0
								9	0
+									5

$$= 4\cdot 10^9 + 2\cdot 10^8 + 9\cdot 10^7 + 4\cdot 10^6 + 9\cdot 10^5 + 6\cdot 10^4 + 7\cdot 10^3 + 2\cdot 10^2 + 9\cdot 10^1 + 5\cdot 10^0$$

$$= d_9\cdot 10^9 + d_8\cdot 10^8 + d_7\cdot 10^7 + d_6\cdot 10^6 + d_5\cdot 10^5 + d_4\cdot 10^4 + d_3\cdot 10^3 + d_2\cdot 10^2 + d_1\cdot 10^1 + d_0\cdot 1$$

$$= \sum_{i=0}^{9} d_i \cdot 10^i$$

where $d_9 = 4$, $d_8 = 2$, $d_7 = 9$, $d_6 = 4$, $d_5 = 9$, $d_4 = 6$, $d_3 = 7$, $d_2 = 2$, $d_1 = 9$, and $d_0 = 5$.

□ The number of digits in a numeric value is the total number of digits used to express the exact written value of the number in decimal form:

Example 5: The number 654321 is a 6-digit number since it is written using six digits with

position 5 position 4 position 3 position 2 position 1 position 0

6 5 4 3 2 1

$= 600{,}000 + 50{,}000 + 4{,}000 + 300 + 20 + 1$

$= 6 \cdot 10^5 + 5 \cdot 10^4 + 4 \cdot 10^3 + 3 \cdot 10^2 + 2 \cdot 10^1 + 1 \cdot 10^0$

This number means:

six	hundred	thousands
five	ten	thousands
four		thousands
three		hundreds
two		tens
one		one

☐ One of the major themes we will return to over and over again in this class is that digital computing (and hence MATLAB) is intimately connected to the representation of numbers. In our previous examples, we see some very interesting patterns that emerge. Let's revisit all of these examples to investigate further.

Positional Interpretation of nonnegative (unsigned) decimal integers												
Position #	position 9	position 8	position 7	position 6	position 5	position 4	position 3	position 2	position 1	position 0		
English name	billions	hundred millions	ten millions	millions	hunrded thousands	ten thousands	thousands	hundreds	tens	ones		
Integer representation	1,000,000,000	100,000,000	10,000,000	1,000,000	100,000s	10,000s	1,000s	100s	10s	1s		
Power of base 10	10^9	10^8	10^7	10^6	10^5	10^4	10^3	10^2	10^1	10^0	Value	Number of digits
Example 1									**1**	**5**	15	n = 2
Example 2								**2**	**5**	**5**	255	n = 3
Example 3						**6**	**5**	**5**	**3**	**5**	65,535	n = 5
Example 4	**4**	**2**	**9**	**4**	**9**	**6**	**7**	**2**	**9**	**5**	4,294,967,295	n = 9
Example 5					**6**	**5**	**4**	**3**	**2**	**1**	654,321	n = 6

☐ Looking at the table above, consider the following questions:

- What does we mean when we say that a nonnegative integer is an n-digit decimal integer?
- How is the position number related to the power of base 10?
- How is the number of digits in a number related to the position numbers illustrated below?
- What do we mean when we say a "decimal integer"?

General form of nonnegative (unsigned) decimal integers

- Looking over our previous examples, we may notice a clear pattern in our representations of nonnegative (unsigned) decimal integers.

- To specify an $(n+1)$-digit unsigned decimal integer y we write such a number using the general form

$$y = \underset{\text{digit } (n+1)}{\overset{\text{position } n}{d_n}}\ \underset{\text{digit } n}{\overset{\text{position } (n-1)}{d_{n-1}}}\ \underset{\text{digit } (n-1)}{\overset{\text{position } (n-2)}{d_{n-2}}} \cdots \underset{\text{digit } 3}{\overset{\text{position } 2}{d_2}}\ \underset{\text{digit } 2}{\overset{\text{position } 1}{d_1}}\ \underset{\text{digit } 1}{\overset{\text{position } 0}{d_0}}$$

$$= d_n \cdot 10^n + d_{n-1} \cdot 10^{n-1} + \cdots + d_2 \cdot 10^2 + d_1 \cdot 10^1 + d_0 \cdot 10^0$$

$$= \sum_{i=0}^{n} d_i \cdot 10^i$$

- The subscripted index i on each digit d_i represents the position number (not the digit number) of each digit and matches the power of 10 associated with each digit

with each decimal digit $d_i \in D = \{0, 1, 2, \cdots, 9\}$ for all i.

Before we move on, let's make a few remarks about notation.

- When writing an $(n+1)$-digit decimal integer

$$d_n\ d_{n-1}\ \cdots\ d_2\ d_1\ d_0$$

d_n: most significant digit; d_0: least significant digit

we will refer to the right-most digit as the least significant digit and we will call the left-most digit the most significant digit. (We will also adopt this pattern in later lessons when considering floating-point numbers).

- The entire decimal number system is said to have a base of 10. This phrase "base of 10" means that each digit in a number is multiplied by a factor of 10 raised to the power corresponding to that digits position.

- A synonym for the word "base" is the word radix. Thus, the decimal number system has a radix of 10. This alternative terminology comes in useful as we explore other number systems that use a radix (base) not equal to 10.

☐ Finally, in the decimal number system, there is a very special reason why the value of each decimal digit

$$d_i \in D = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

must be between 0 and 9.

☐ Consider the number $y = 607$.

position 2 → position 1 → position 0 →

How many tens are in this number?

☐ If we answer this question using our knowledge of position #s we might be tempted to claim there are zero 10's since

$$y = 607 = 6 \cdot 10^2 + 0 \cdot 10^1 + 7 \cdot 10^0$$

↑ Zero in the tens position

☐ However, if we answer this question using our knowledge of conversational english, we might say there are 60 tens since $60 \times 10 = 600$.

☐ These two different paradigms lead to an important observation about how decimal digits are designed to function:

- each position holds only the "leftover" numbers, stored as digits that cannot be lumped into a higher position value.

- thus, each digit in a particular position cannot take a value greater than 9 since nine is the maximum value that a position can hold before the values can be lumped together and flipped into the next highest position.

☐ When writing very large decimal integers, we might find it natural to group digits into chunks of three digits each starting from the least significant digit and moving left:

$$4294967295 = 4\ 294\ 967\ 295$$
$$= 4,294,967,295$$

This habit follows nicely from our use of commas to delimit every three digits.

The binary number system and unsigned binary integers

- In the decimal number system, the value of each digit might take one of ten different choices with

$$d_i \in D = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

- There is an alternative positional number system that forms the foundation for how modern-day digital computers store and manipulate numbers and characters. This other system is known as the binary number system.

- In the binary system, each digit can take only one of two values: 0 or 1. Moreover, all integers in this system are represented using base 2 expansions.

- Because we are now venturing into a study that includes multiple number systems, we want to be sure to avoid confusion. Thus, we will sometimes put a subscript on a number to indicate the base used to encode the specific representation (see examples 6 - 10).

Example 6: Let's begin by considering a two-digit, nonnegative (unsigned) binary integer

$$y = (11)_2 = \begin{matrix} \text{digit 2} & \text{digit 1} \\ \downarrow & \downarrow \\ 1 & 1 \\ \uparrow & \uparrow \\ \text{position 1} & \text{position 0} \end{matrix}$$

↑ this subscript symbolizes the fact that our representation is written in base 2

$$= \boxed{1} \cdot 2^{\boxed{1}} + \boxed{1} \cdot 2^{\boxed{0}}$$

(position 1 → exponent; position 0 → exponent; digit 2 → first 1; digit 1 → second 1)

$$= 2 + 1$$

$$= (3)_{10}$$

↑ this subscript represents the fact that the final number 3 is written as a decimal integ

How to count digits and positions in binary system?

☐ Notice that the method we use to count digits and positions in our binary syste is the same as in our decimal system. However, two major differences are that the position number in a binary integer stores the power of 2 associated with each position and the value of each digit is limited within an "alphabet" of two symbols 0 or 1.

Example 7: Let's take a look at a four-digit nonnegative (unsigned) binary integer

$y = 1\ 0\ 0\ 1$ (digit 4 → 1 ← position 3; digit 3 → 0 ← position 2; digit 2 → 0 ← position 1; digit 1 → 1 ← position 0)

$= (1\ 0\ 0\ 1)_{(2)}$

↖ this subscript clarifies that our integer is written w/ respect to radix value of 2

$= \boxed{1} \cdot 2^{\boxed{3}} + \boxed{0} \cdot 2^{\boxed{2}} + \boxed{0} \cdot 2^{\boxed{1}} + \boxed{1} \cdot 2^{\boxed{0}}$

(position 3 ↓ 3; position 2 ↓ 2; position 1 ↓ 1; position 0 ↓ 0)

(digit 4 ↑ 1; digit 3 ↑ 0; digit 2 ↑ 0; digit 1 ↑ 1)

$= 8 + 1$

$= (9)_{(10)}$

↖ this subscript indicates that our final number 9 is written as a decimal integer.

Example 8: Let's extend example 7 and write out all possible four-digit, unsigned binary integers along with the corresponding decimal representations, as seen in the table below.

4-Digit Binary	Decimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	10
1011	11
1100	12
1101	13
1110	14
1111	15

← let's check the highlighted row

Check: $(1101)_2 = (13)_{10}$

$y = (1\ 1\ 0\ 1)_2$

$= 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$

$= 8 + 4 + 0 + 1 = (13)_{10}$ ✓

Example 9: Now let's analyze a 7-digit, nonnegative (unsigned) binary integer

$$y = (1\ 1\ 0\ 0\ 1\ 0\ 0)_2$$

digit 7 → 1 ← position 6
digit 6 → 1 ← position 5
digit 5 → 0 ← position 4
digit 4 → 0 ← position 3
digit 3 → 1 ← position 2
digit 2 → 0 ← position 1
digit 1 → 0 ← position 0

$$= 1 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2$$

$$= 64 + 32 + 4$$

$$= (100)_{10}$$

We are now in a positon to begin to find a general form for binary integers:

$$y = (1\ 1\ 0\ 0\ 1\ 0\ 0)_2 = b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0$$

where $b_6 = 1$, $b_5 = 1$, $b_4 = 0$, $b_3 = 0$, $b_2 = 1$, $b_1 = 0$, and $b_0 = 0$.

Example 10: Finally, let's look at an 8-digit binary integer given below

$$y = (\underset{\text{digit 8}\to\ \uparrow\text{ position 7}}{1}\ \underset{\text{digit 7}\to\ \uparrow\text{ position 6}}{1}\ \underset{\text{digit 6}\to\ \uparrow\text{ position 5}}{1}\ \underset{\text{digit 5}\to\ \uparrow\text{ position 4}}{1}\ \underset{\text{digit 4}\to\ \uparrow\text{ position 3}}{1}\ \underset{\text{digit 3}\to\ \uparrow\text{ position 2}}{1}\ \underset{\text{digit 2}\to\ \uparrow\text{ position 1}}{1}\ \underset{\text{digit 1}\to\ \uparrow\text{ position 0}}{1})_2$$

$$= 1\cdot 2^7 + 1\cdot 2^6 + 1\cdot 2^5 + 1\cdot 2^4 + 1\cdot 2^3 + 1\cdot 2^2 + 1\cdot 2^1 + 1\cdot 2^0$$

$$= 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1$$

$$= (255)_{10}$$

☐ As we see in Example 10, the larger the number of binary digits that we write, the more important it is that we can quickly find powers of 2. Below is a list of many powers of 2:

Powers of 2	
n	2^n = Decimal
0	$2^0 = 1$
1	$2^1 = 2$
2	$2^2 = 4$
3	$2^3 = 8$
4	$2^4 = 16$
5	$2^5 = 32$
6	$2^6 = 64$
7	$2^7 = 128$
8	$2^8 = 256$
9	$2^9 = 512$
10	$2^{10} = 1{,}024$
11	$2^{11} = 2{,}048$
12	$2^{12} = 4{,}096$
13	$2^{13} = 8{,}192$
14	$2^{14} = 16{,}384$
15	$2^{15} = 32{,}768$
16	$2^{16} = 65{,}536$
17	$2^{17} = 131{,}072$
18	$2^{18} = 262{,}144$
19	$2^{19} = 524{,}288$
20	$2^{20} = 1{,}048{,}576$
21	$2^{21} = 2{,}097{,}152$
22	$2^{22} = 4{,}194{,}304$
23	$2^{23} = 8{,}388{,}608$
24	$2^{24} = 16{,}777{,}216$
25	$2^{25} = 33{,}554{,}432$
26	$2^{26} = 67{,}108{,}864$
27	$2^{27} = 134{,}217{,}728$
28	$2^{28} = 268{,}435{,}456$
29	$2^{29} = 536{,}870{,}912$
30	$2^{30} = 1{,}073{,}741{,}824$
31	$2^{31} = 2{,}147{,}483{,}648$
32	$2^{32} = 4{,}294{,}967{,}296$

☐ Notice that all the powers n are written as decimal integers.

General form of nonnegative (unsigned) binary integers

- Looking back over examples 5 – 10, we may again notice a clear pattern in our representation of nonnegative (unsigned) binary integers.

- To specify an (m+1)-digit unsigned binary integer y we write this number using the general form

$$y = b_m \; b_{m-1} \; b_{m-2} \cdots b_2 \; b_1 \; b_0$$

digit (m+1) → position m; digit m → position (m-1); digit (m-1) → position (m-2); digit 3 → position 2; digit 2 → position 1; digit 1 → position 0

$$= b_m \cdot 2^m + b_{m-1} \cdot 2^{m-1} + \cdots + b_2 \cdot 2^2 + b_1 \cdot 2^1 + b_0 \cdot$$

$$= \sum_{i=0}^{m} b_i \cdot 2^i$$

where each binary digit $b_i \in \{0, 1\}$ for all values of i.

Before we move on, let's make a few remarks about notation

- When writing an $(m+1)$-digit nonnegative binary integer given by

$$b_m\, b_{m-1} \cdots b_2\, b_1\, b_0$$

each digit b_i is known as a **bit**, which is a contraction of the two words **bi**nary digi**t**

- As we saw in our previous example, each bit takes one of two values: either 0 or 1. In terms of the "hardware in a computer, these two values are stored as low or high voltages.

EXAMPLE 3.10 1/2

Consider the real-world problem: How can we design a machine to perform calculations between numbers? The first generation of scientists to focus on this problem realized that all data could be encoded using a two letter alphabet,$\{0, 1\}$. The symbol 0 is called logic 0 and the symbol 1 is known as logic 1 (for more about this, Google search"Boolean Algebra"). Claude Shannon, the father of Information Theory and the first person to draw connections between formal mathematical logic and electric circuits, realized that electricity could be used encode the value of a two letter alphabet into a machine. In particular, given a high voltage H, and a Low voltage L, an explicit relation could be established between the fundamental unit of encoding and physical measurements of electric current. For example, we could define a relation, known as the active high relation where

$$A_H = \{(0, L), (1, H)\}.$$

This corresponds with a digital computer in which high voltages stored in the computer's hardware are interpreted as logic 1 while low voltages are interpreted as logic 0. On the other hand, the relation known as active low has the opposite orientation:

$$A_L = \{(0, H), (1, L)\}.$$

In this case, high voltages correspond to logic 0 while low voltages to logic 1.

It is worth noting that the actual physical values of H and L vary greatly depending on the fabrication process of the digital hardware. For example, the Texas Instrument TTL (Transistor-Transistor Logic) family has a H value of 3.3 Volts and a L value of 0.5 Volts.

Converting from binary to decimal representation

As we saw in Examples 3.6 - 3.10, to convert a given nonnegative binary integer into a decimal integer, we simply multiply each binary digit by the power of 2 as indicated by that digits position and then add the results. This process of converting a $(m+1)-$digit binary representation of a nonnegative integers $N \geq 0$ in the form

$$N = \left(b_m b_{m-1} ... b_2 b_1 b_0 \right)_2$$

into the corresponding $(n+1)-$digit decimal representation of the the same number given by

$$N = \left(d_n d_{n-1} ... d_2 d_1 d_0 \right)_{10}$$

is one example of a large class of techniques known as radix conversation algorithms.

The MATLAB Environment stores all unsigned integer variables in the workspace in binary representation. However, when we view the value of said variables in the command window, MATLAB displays these value(s) using the decimal representation of these numbers. To do so, MATLAB converts the stored values into decimal form using the proper radix conversion algorithm. Thus, the process of converting binary representations into decimal representations is known as an output problem since it is used to display meaningful output to MATLAB users.

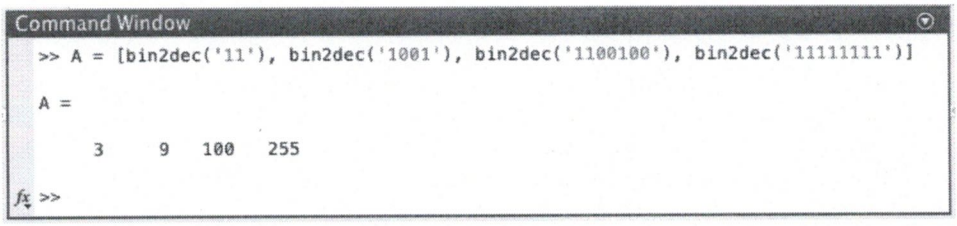
Luckily, MATLAB has a built-in function, called `bin2dec`, that does these type of calculations quickly. The general syntax to call this function is as follows:

```
bin2dec('binary_string')
```

This function interprets the `binary_string` as text that represents a binary number and expects this input to be within the left and right single quotation mark ' (indicating the input is a string). When we press Enter , the output will be the equivalent decimal number. The input `binary_string` should be an unsigned binary representation of a nonnegative integer N whose value is bounded as follows

$$0 \leq N < 2^{52} = 4,503,599,627,370,496$$

where `flintmax` is the largest consecutive integer in floating-point format (to be discussed in Lesson 5). Let's take a look how to convert some of the binary integers from examples 3.6 - 3.10.



Lesson 3, Figure 1: Using `bin2dec` to find decimal representation

Converting from decimal to binary representation

The process of converting a given nonnegative decimal integer into a binary representation is a bit more nuanced. In this case, we begin with a $(n+1)-$digit decimal representation of the a nonnegative integer given by

$$N = \left(d_n d_{n-1} ... d_2 d_1 d_0 \right)_{10}$$

and seek to find it's corresponding unsigned $(m+1)-$digit binary representation

$$N = \left(b_m b_{m-1} ... b_2 b_1 b_0 \right)_2$$

We begin our discussion with an illustrative example.

EXAMPLE 3.11

Let's convert the unsigned decimal integer $N = (11)_{10}$ into binary form. To do so, we start with a useful observation. Every time we divide a number by 2, we produce both a quotient and a remainder value. Because of the binary nature of the number 2, the remainder will always be equal to either 0 or 1. Let's take a look at repeatedly dividing $N = (11)_{10}$ by 2.

Step i	Division	=	Quotient N_i	Remainder: b_{i-1}	Position
1	$\frac{11}{2}$	=	5	1	0
2	$\frac{5}{2}$	=	2	1	1
3	$\frac{2}{2}$	=	1	0	2
4	$\frac{1}{2}$	=	0	1	3

The sequence of remainder digits $b_3 b_2 b_1 b_0$ now encodes our binary representation. In other words, we claim that

$$(11)_{10} = (1101)_2$$

which can be quickly confirmed by carrying out the corresponding arithmetic.

The conversion algorithm illustrated in the example above is based on a useful encoding of division. Let's rewrite the information contained in our table above in a different form:

Step 0 :	Initialize	$N_0 = N$
Step 1 :	$11 = 5 \cdot 2 + 1,$	$N_0 = N_1 \cdot 2 + b_0$
Step 2 :	$5 = 2 \cdot 2 + 1,$	$N_1 = N_2 \cdot 2 + b_1$
Step 3 :	$2 = 1 \cdot 2 + 0,$	$N_2 = N_3 \cdot 2 + b_2$
Step 4 :	$1 = 0 \cdot 2 + 1,$	$N_3 = N_4 \cdot 2 + b_3$

This sequence of repeated divisions produces an equivalent binary encoding that proceeds in the opposite direction. In particular, notice that we can generalize to combine our first two results

Step 1:	$N_0 = N_1 \cdot 2 + b_0$
Step 2:	$N_1 = N_2 \cdot 2 + b_1$

together to see

$$\begin{aligned} N &= N_1 \cdot 2 + b_0 \\ &= (N_2 \cdot 2 + b_1) \cdot 2 + b_0 \\ &= N_2 \cdot 2^2 + b_1 \cdot 2^1 + b_0 \cdot 2^0 \end{aligned}$$

Next, we can use our information from step 3 to continue this process.

$$\begin{aligned} N &= N_2 \cdot 2^2 + b_1 \cdot 2^1 + b_0 \cdot 2^0 \\ &= (N_3 \cdot 2 + b_2) \cdot 2^2 + b_1 \cdot 2^1 + b_0 \cdot 2^0 \\ &= N_3 \cdot 2^3 + b_2 \cdot 2^2 + b_1 \cdot 2^1 + b_0 \cdot 2^0 \end{aligned}$$

As long as our original unsigned integer $N \geq 2$, the sequence of quotients

$$N > N_1 > N_2 > \cdots > N_m > N_{m+1}$$

must eventually produce a quotient $N_{m+1} = 1$. At this stage, the remainder b_m is either 0 or 1. This results in the $(m+1)-$digit binary representation of our original unsigned integer

$$\begin{aligned} N &= 1 \cdot 2^m + b_{m-1} \cdot 2^{m-1} + \cdots + b_2 \cdot 2^2 + b_1 \cdot 2^1 + b_0 \\ &= \left(b_m b_{m-1} ... b_2 b_1 b_0 \right)_2 \end{aligned}$$

EXAMPLE 3.12

Let's convert the unsigned decimal integer $N = (145)_{10}$ into binary form. We repeat our algorithm demonstrated above, step-by-step to produce our desired binary representation.

Step i	Division	=	Quotient N_i	Remainder: b_{i-1}	Position
1	$\frac{145}{2}$	=	72	1	0
2	$\frac{72}{2}$	=	36	0	1
3	$\frac{36}{2}$	=	18	0	2
4	$\frac{18}{2}$	=	9	0	3
5	$\frac{9}{2}$	=	4	1	4
6	$\frac{4}{2}$	=	2	0	5
7	$\frac{2}{2}$	=	1	0	6
8	$\frac{1}{2}$	=	0	1	7

The sequence of remainder digits encodes our binary representation with

$$(145)_{10} = (10010001)_2$$

The MATLAB Environment displays all unsigned integer variables in the Command Window in decimal representation [1]. However, when we store these value in the workspace, MATLAB encodes these numbers as unsigned binary integers of the appropriate size. Thus, the process of converting decimal representations into binary representations is known as an input problem since it is used to encode meaningful input.

[1] If we are being fully honest, all data displayed in the command window is represented using characters on the screen. Thus, MATLAB actively converts the unsigned binary representations into a corresponding representation where each digit is represented in Unicode. This conversion process, while interesting, is beyond the scope of this class. For our purposes, we focus on the mathematical interpretations since our main use of MATLAB will be to execute numerical computations.

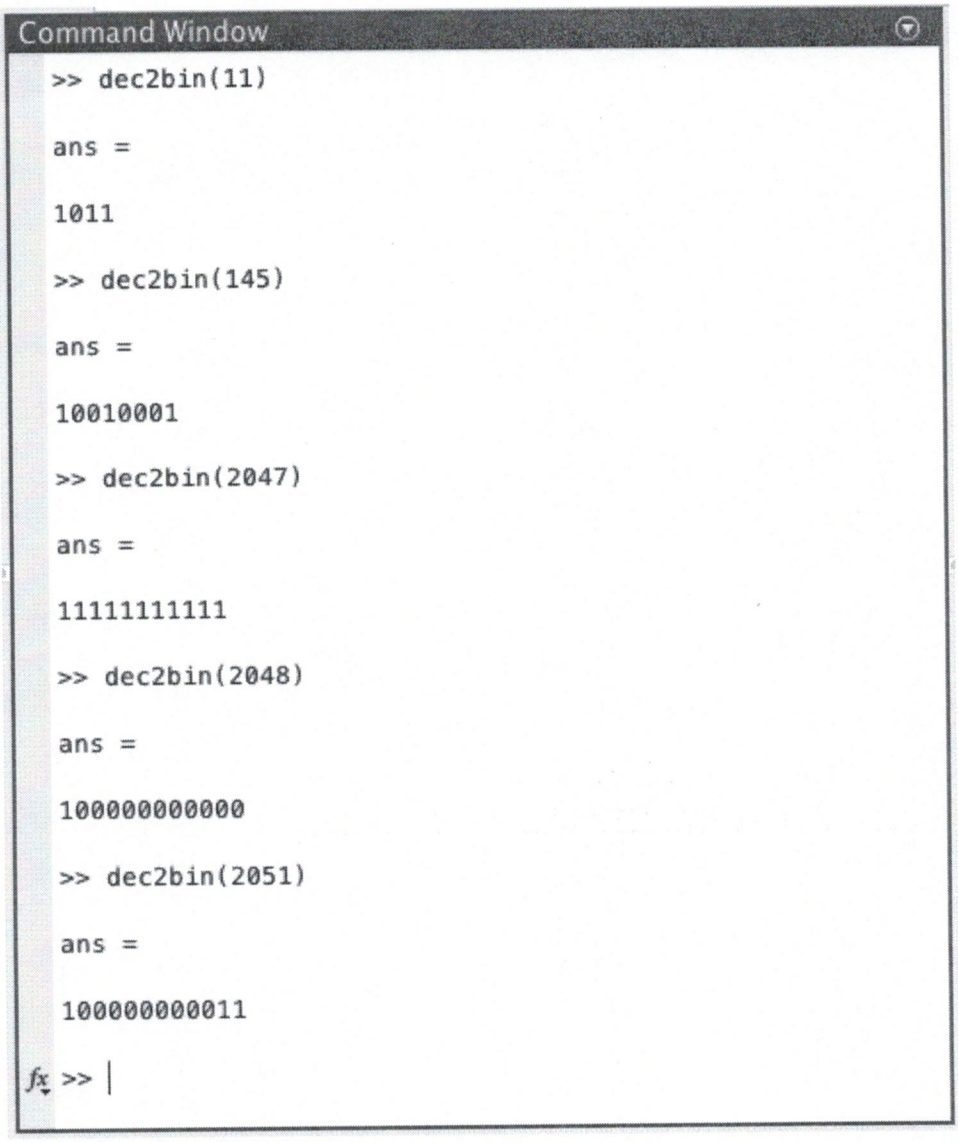
Just as before, MATLAB provides a built-in function, called `dec2bin`, that will convert decimal nonnegative integers into binary representation. The general syntax to call this function is as follows:

```
binary_string = dec2bin(N)
```

This function interprets the input `N` as a nonnegative with value

$$0 \leq N < \texttt{flintmax} = 2^{53} = 9,007,199,254,740,992.$$

The output `biary_string` is a binary representation of N, represented as a string. Let's take a look how to use this function in action.



Lesson 3, Figure 2: Using `dec2bin` to find binary representation

The hexadecimal number system for unsigned integers

One of the major themes of this class is that, on a digital computer, we represent, encode, and store all forms of data using binary codes. For the last part of this lesson, we explore a compact mechanism to represent raw binary data, known as hexadecimal notation. While binary encoding systems are extremely convenient for storing data on a digital computer, these data encodings are much more cumbersome for human beings to work with. More specifically, because each bit (binary digit) encodes either a 0 or a 1, we end up needed to write out long, tedious strings of binary digits to encode relatively simple data. With this in mind, professional programmers who spend many hours a day working with actual raw data encoded on a computer often want to represent such data in a more compact form. For this purpose, we adopt *hexadecimal notation* which uses a base 16 number system.

In hexadecimal notation, we group all binary digits into sets of four bits. Each group of four bits is called a *nibble* and all possible combinations of four binary digits is given a special symbol in our hexadecimal alphabet, as seen below:

4-bit binary niblle	lowercase hexadecimal	Uppercase Hexadecimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	a	A
1011	b	B
1100	c	C
1101	d	D
1110	e	E
1111	f	F

This notation is particularly convenient for expressing strings of binary digits because no tedious conversion between forms is needed. This new method for representing raw binary data suggests a useful convention: we should get in the habit of writing all binary data using nibbles (chunks of four-bit binary numbers) [2]. Let's take a look how we can use this information in practice.

[2]This is analogous to the idea of chunking large decimal numbers into three decimal digits separated by a comma.

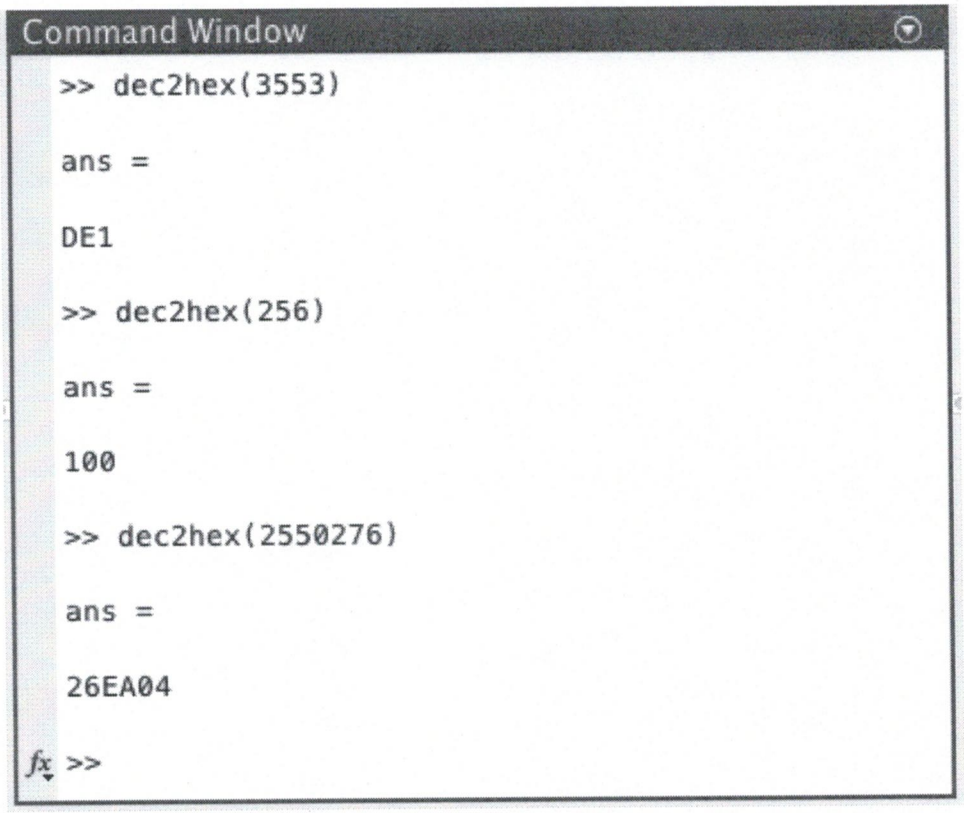
MATLAB does provide a built-in function to convert decimal numbers into hexadecimal form. This `dec2hex` function converts an unsigned decimal integers into it's hexadecimal representation. The general syntax to call this function is as follows:

```
string = dec2hex(N)
```

The input `N` must be a nonnegative integer value with

$$0 \leq N < \texttt{flintmax} = 2^{52}.$$

The output `string` is a hexadecimal representation of N, represented as a string. Let's take a look how to use this function in action.



Lesson 3, Figure 3: Using `dec2hex` to find hexadecimal representations

While there is no built-in function that converts binary strings into hexadecimal notation, the MATLAB community provides a User-built function to us for free via MATLAB File Exchange. To get access to this function, google search the phrase "bin2hex File Exchange MATLAB" and click on the top result.

MATLAB Data Classes for Unsigned Integers

After our brief introduction to number systems used to represent nonnegative integers, let's explore how we can use MATLAB to store such numbers. As we will see, MATLAB's default is to store all numerical data as a double precision floating point number (we will discuss this in much greater detail in Lesson 5). However, floating point representations tend to be binary codes that are quite complex. Thus,

we begin our work to develop intuition for the first steps of this representation which is to explore the different classes that MATLAB provides to store unsigned integers.

Before we do so, let's explore one last note on notation. In 1964, the designers of the IBM System/360 mainframe computer established a related convention of using groups of 8−bits as the basic unit of addressable computer storage. These computer architects referred to each collection of 8 bits as a *byte*, a convention that is still used today. With this in mind, we usually think of the raw binary data that is stored in memory as being grouped into *words*. Each word consists of two or more adjacent bytes. Moreover, these words are almost always addressed and manipulated collectively as a single unit of memory. Thus, we see the following hierarchy in memory:

$$\text{bit} \longrightarrow \text{byte} \longrightarrow \text{word} \longrightarrow \text{class}$$

There are four such classes that we can use to store unsigned integers including

`uint8`, `uint16`, `uint32`, and `uint64`

The prefix `uint` stands for unsigned `integer`. The number that ends each of these classes represents the number of bits dedicated to each class. For example, suppose we store the unsigned decimal integer $N = (53)_{10}$ using the `uint8` class. When we do so, MATLAB will allocate 8−bits in memory to store this number. On the other hand, if we store $N = (53)_{10}$ using the `uint64` class, MATLAB will dedicate 64 bits to store this number. Let's take a look at an example below.

```
Command Window
>> n08 = uint8(53)

n08 =

   53

>> n16 = uint16(53)

n16 =

    53

>> n32 = uint32(53)

n32 =

         53

>> n64 = uint64(53)

n64 =

                  53

>> whos
  Name      Size            Bytes  Class     Attributes

  n08       1x1                 1  uint8
  n16       1x1                 2  uint16
  n32       1x1                 4  uint32
  n64       1x1                 8  uint64

fx >>
```

Lesson 3, Figure 4: Using `uint` classes to store unsigned integers

When we use the `whos` command to view more information about each variable, we notice an interesting feature of each unsigned integer class. Specifically, while

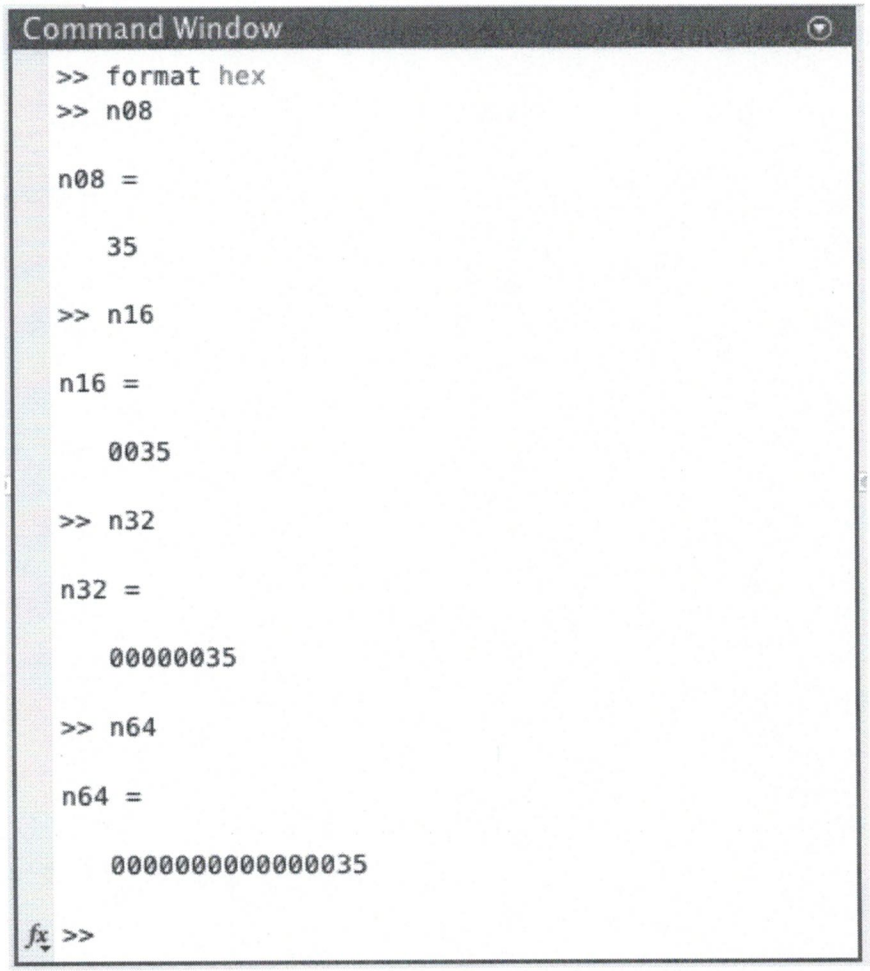
the value of each variable is identical, the amount of memory used to store each variable is not the same. The `uint8` class requires one byte (8 bits) while the `uint64` class requires 8 bytes of data to store the exact same number.

This example leads to a subtle yet powerful realization that when we choose a class to store data, we effect the method MATLAB writes that data into memory. Let's discover more about this. In MATLAB, we can control the format of the displayed output in the command window. There are two types of ways we can control Command Window output formats including:

1. Numeric formats that affect only how numbers appear in the command window. When controlling numeric formats, we change only how numbers appear in the Command Window but we do NOT effect the style display of the output nor how MATLAB stores or computes the numbers in the workspace.

2. Style display formats that affect the "style" of the output displayed in the Command Window.

To control the format of output in the Command Window, we use the `format` command. For example, if we type the command `format hex` into the Command Window and press Enter, then all of our variables will be displayed in hexadecimal format. Let's take a look at why this matters.



Lesson 3, Figure 5: Using `format hex` to explore `uint` classes

As we already saw, the decimal value of all four `n08, n16, n32, n64` variables is identical. However, the actual binary codes used to store these variables in memory is not. By using the `format hex` command, we get to view the hexadecimal representation of the raw binary data stored in memory. Let's take a look at another example where these differences come into play. In this case, we will try to store the unsigned integer

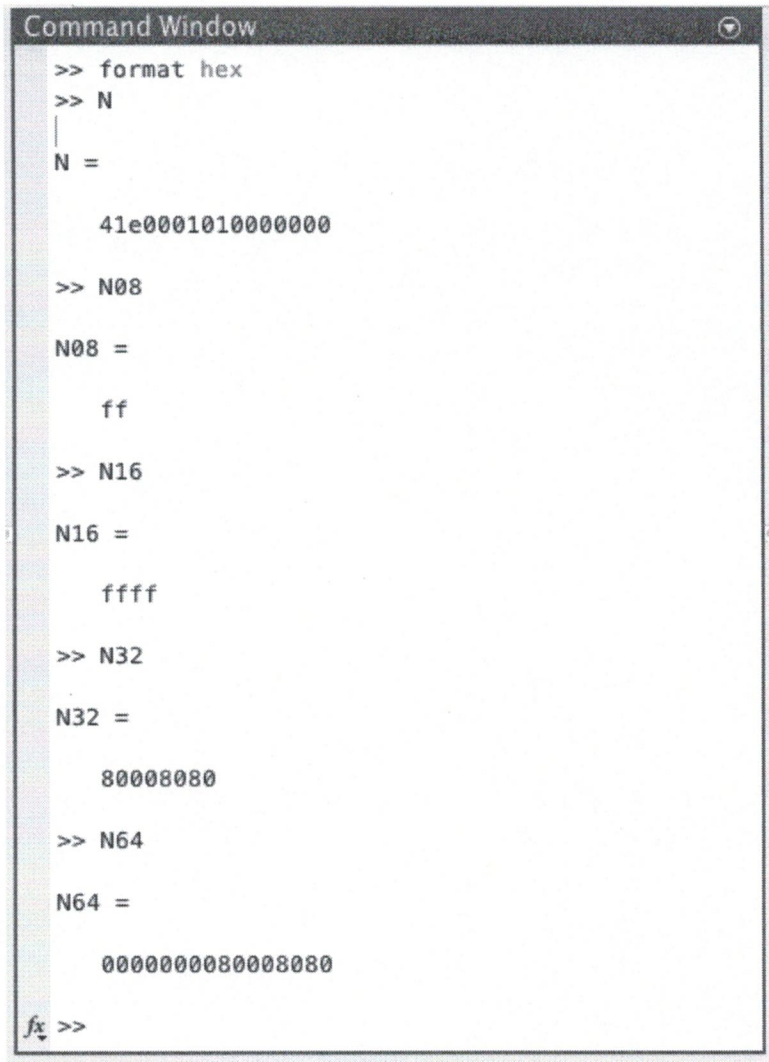
$$N = 2^{31} + 2^{15} + 2^7 = 2,147,516,544$$

First, let's change the output format for the Command Window back into the default form by typing `format short`.

```
Command Window
>> format short
>> N = 2^31 + 2^15 + 2^7

N =

   2.1475e+09

>> N08 = uint8(2^31 + 2^15 + 2^7);
>> N16 = uint16(2^31 + 2^15 + 2^7);
>> N32 = uint32(2^31 + 2^15 + 2^7);
>> N64 = uint64(2^31 + 2^15 + 2^7);
>> N64

N64 =

           2147516544

>> N32

N32 =

  2147516544

>> N16

N16 =

  65535

>> N08

N08 =

  255

fx >>
```

Lesson 3, Figure 6: Using `uint` classes for large unsigned integers

Notice that some very strange results are displayed. In particular, the default variable assignment `N = 2^31 + 2^15 + 2^7` results in output that does not look like an integer. As we will see in Lesson 5, this is due to the fact that unless told to do otherwise, MATLAB stores numeric variables as double precision floating point data. On the other hand, if we store this exact same number using all four of our `uint` classes, the resulting output is not identical. We can explore more about this by viewing the value of each variable in `hex` format.



Lesson 3, Figure 7: Using `format hex` to explore values of large unsigned integers

In this example, the `uint08` and `uint16` classes produce garbage data. This error results from *overflow* in which we try to store data whose exact value is larger than the largest possible unsigned integer in these two classes. In fact, each unsigned integer class can store a range of values. The upper bound on each class is directly related to the number of bits assigned to each piece of data. Let's create a table of values for the range of values that are possible for each `uint` class. This table suggests natural choices which unsigned integer class we should use to store nonnegative integer values, assuming that we have a good idea of how large our integers might be.

- A general m-bit unsigned binary integer $(y)_2$ can be used to represent any unsigned decimal integer $(y)_{10} \in \mathbb{Z}$ within a range of values with $0 \leq (y)_{10} \leq 2^m - 1$

- MATLAB provides four unsigned integer classes to store nonnegative binary integers, as shown below.

MATLAB Class	Number of bits	Number of bytes	General form for binary integer	Lower bound (decimal)	Range of values (decimal)	Upper bound (decimal)	MATLAB Function to store integer
Unsigned 8-bit integer	8	1	$(y)_2 = b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0$	0	$0 \leq (y)_{10} \leq (2^8 - 1)$	255	`uint8`
Unsigned 16-bit integer	16	2	$(y)_2 = b_{15} b_{14} b_{13} b_{12} b_{11} b_{10} b_9 b_8 b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0$	0	$0 \leq (y)_{10} \leq (2^{16} - 1)$	65,535	`uint16`
Unsigned 32-bit integer	32	4	$(y)_2 = b_{31} b_{30} b_{29} \ldots b_2 b_1 b_0$	0	$0 \leq (y)_{10} \leq (2^{32} - 1)$	4,294,967,295	`uint32`
Unsigned 64-bit integer	64	8	$(y)_2 = b_{63} b_{62} b_{61} \ldots b_2 b_1 b_0$	0	$0 \leq (y)_{10} \leq (2^{64} - 1)$	18,446,744,073,709,500,000	`uint64`