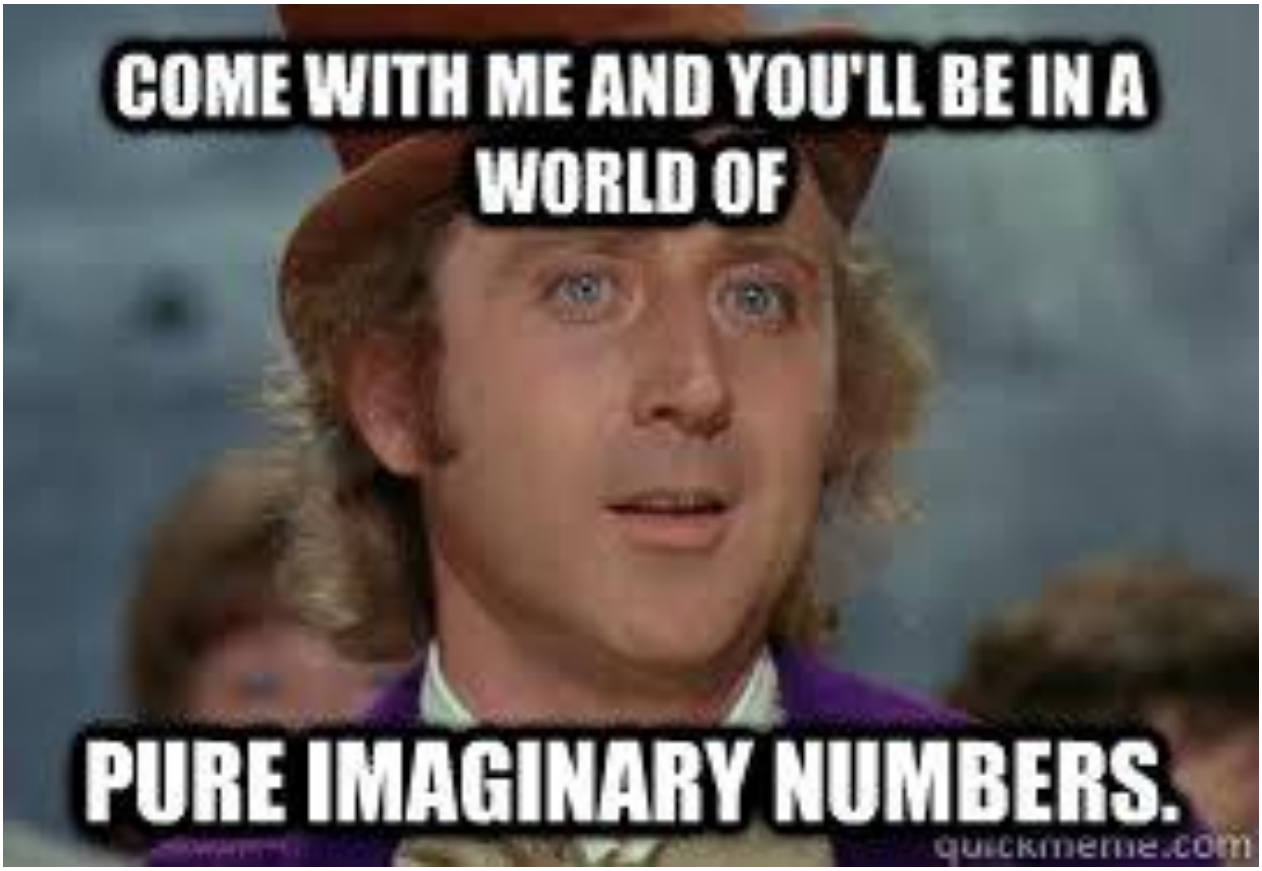
Introduction to Numbers

By Alexander Wang

42 Silicon Valley



FOREWARD

Welcome, at the suggestion of my editor, Jo Chang and the future editor who read this, I have rewritten the initial pdf into their own more “specialized” categories. And as a result, and the hope of the author here that it will result in significantly less information needed (read page length) for the same amount of information. There were a number of reasons for this restructuring of the pdfs as the author had initially thought and confirmed by the first editor:

1. Not feasible for you guys to be able to do the project in one day.
2. Too much initial information required before you can begin.
3. The overall page length of the initial PDF was starting to approach unwieldiness if not already there.
4. Continuing with 3, the first draft of the PDF acted more like a reference book rather than a tutorial.

Primarily because of number 4, which stemmed as a result of the initial 3 reasons, the author has reduced the length of the so-called “reference” book by condensing and separating it into a number of “sub-textbooks” like the one you are currently reading.

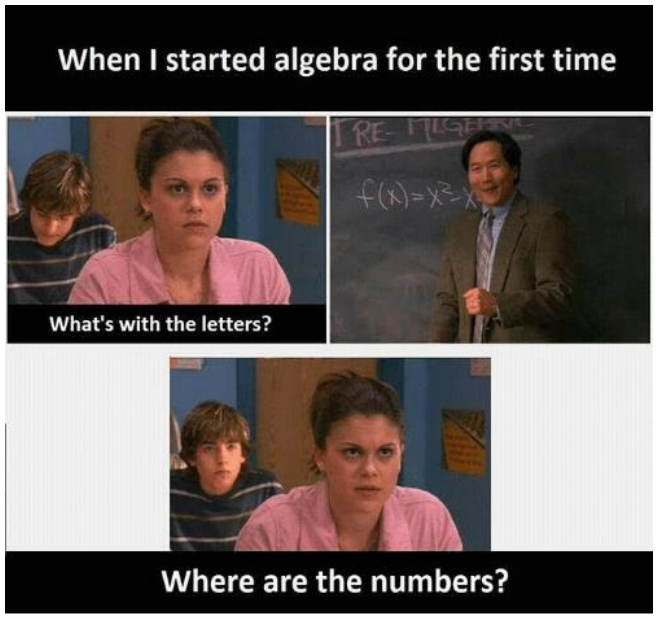
Now what is this particular “textbook” about? In a word, numbers. Based on your initial experience with Python, you probably had everything related to numbers done for you and you didn’t need to understand the underlying principles of them or at least “why” it’s the way they are.

This textbook and the subsequent exercises related to it will remedy that very quickly.

It is the author's expectations of at least the following prerequisite information before attempting to read this "textbook":

- Know how to add, multiply, and divide, and potentially modulus (modulus will be covered regardless).
- Know basic Algebra, currently taking an Algebra 1-2 course or equivalent, higher math classes will obviously make this easier but otherwise including but not limited to:
    - Algebraic expressions
    - Functions
    - Variables
    - Solving for variables
    - Properties of equality and inequalities
    - Exponential and logarithms
    - Radical expressions

Let's begin.



Chapter 1: Some Preliminary Information

Introduction:

Welcome to this lovely review on algebra. For a good chunk of the people who are reading this, you can skip this chapter as it will be covering some relatively simple things. This is not meant to replace a full on textbook for Algebra but rather a “simple” primer. I say simple, but my editors will probably disagree with me and this will probably be scrapped in later editions but whatever I’m keeping it (go away Jo).

Now what will we be covering? A lot of things, but I will be hand-waving *a lot* of the explanations and would like to encourage you to read *Elements of Algebra* by Euler (a more condensed but otherwise *very good* explanation of the so-called “real” Algebra) as this was meant to be a quick and simple explanation of concepts.

Bases:

A base is a number system with a set of digits. Some common ones are:

- (Decimal) Base-10 – the ones we traditionally use.
- (Binary) Base-2 – the ones computers use.
- (Hexadecimal) Base-16 – The ones we use when we don't want to use binary numbers or base-10.
- Base-N – Where n is a number, and represents how many digits the base has.

We will be discussing binary and hexadecimal in-depth in the next two chapters. But for now here is 2 factoids to remember that I should prove to you but for now just accept it as fact:

1. Any base can be converted to any other base
2. All numbers (in particular the whole numbers) are representable in any given base.

Note: If you want to see the formal proof for those 2 facts above please see Appendix A. However, ideally, you would have taken at least a Calculus class before reading the proof.

Modulus:

While the formal definition of modulus is actually quite extensive and requires some prior knowledge we will be skipping that entirely and going straight to the simplified informal and horribly flawed definition (where math professors, teachers, tutors, and the like will hate you and probably want to strangle you for skipping steps):

"The final remainder (whole number) that is left when you can't divide a number by another number anymore"

The symbol for this is traditionally (mathematics first then programming):

$$a \; mod \; b = remainder$$
$$a \; \% \; b = remainder$$

Examples of modulus and outcomes:

$$5 \; \% \; 3 = 2$$
$$5 \; \% \; 5 = 0$$
$$5 \; mod \; 5 = 0$$
$$10 \; \% \; 2 = 0$$
$$10 \; \% \; 3 = 1$$

The explanation for these (starting from the top):

1. Can't evenly divide 5 by 3, and the remainder is 2
2. 5 can be divided into 5, so remainder is 0.
3. Equivalent to 2.
4. 2 can evenly divide into 10, and there is no remainder so 0
5. 3 can go into ten 3x, so the remainder is 1.

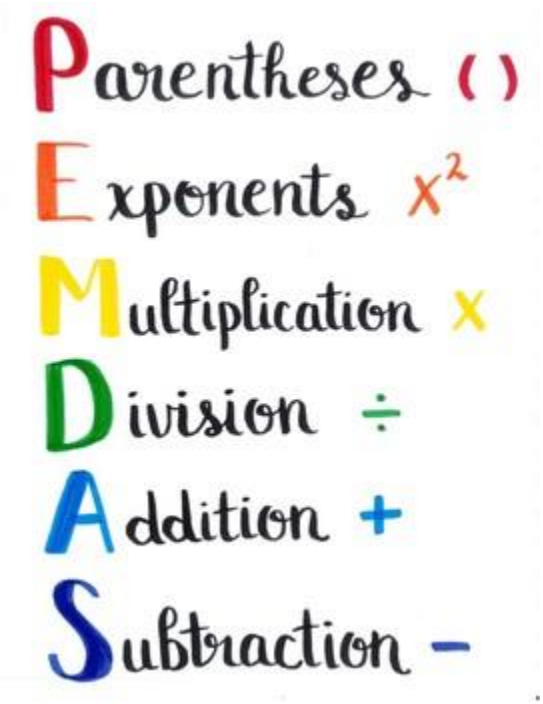
Powers:

So, powers, do you have the power? Just kidding. This is a very simple concept. Just think of it as multiplying the number by itself a certain number of times. Here are some examples:

$$2^5 = 2 * 2 * 2 * 2 * 2 = 32$$
$$5^2 = 5 * 5 = 25$$
$$3^9 = 3 * 3 * 3 * 3 * 3 * 3 * 3 * 3 * 3 = 19683$$

Now let's make it slightly more complicated. Remember **PEMDAS**?

If not here it is again in picture format:



Why did I show that picture? For this example:

$$(1 + 2^2) = \big(1 + (2 * 2)\big) = (1 + 4) = 5$$

Now that is simple. So let's make it a little bit harder, what about adding powers together:

$$2^2 + 2^2 = 2 * 2 + 2 * 2 = 4 + 4 = 8$$

Now for some rules for powers as reference (there is 1 other rule but these 4 are the most important to know for this textbook):

Product rule:

$$x^m * x^n = x^{m+n} \; for \; any \; x, and \; any \; m \; and \; n > 0$$

Example:

$$4^2 * 4^3 = 4 * 4 * 4 * 4 * 4$$
$$4^{2+3} = 4 * 4 * 4 * 4 * 4$$

Power rule:

$$x^{mn} = x^{m*n}$$

Example:

$$(4^2)^3 = 4^{2*3} = 4^6$$

Zero Rule:

$$x^0 = 1 \; for \; x \neq 0$$

Rule of 1:

$$x^1 = x$$
$$3^1 = 3$$
$$1^m = 1$$
$$1^4 = 1 * 1 * 1 * 1 = 1$$

Chapter 2: Binary

Introduction:

That was a rather short introduction to some math concepts. And now we will go into binary. Binary is how the computer represents everything. Think of our normal number system (subscripts is the base):

$$1 + 2 + 3 + 4 \ldots + 9 = 45_{10}$$

We have 10 digits in our number systems and they are:

$$0, 1, 2, 3, 4, 5, 6, 7, 8, 9$$

And we can do a number of things with them, mainly one of these operations:

$$a + b, a * b, \frac{a}{b}, a \,\%\, b$$

Now, we will be wrapping our head around a different number system. Binary and it only has two digits:

$$0, 1$$

We can do the same thing as everything in our decimal system above. Binary is just a different way of representing numbers. And to keep it simple here is a list of numbers in binary (on the left) and their decimal counterparts on the right:

Binary	Decimal
0	0
1	1
10	2
11	3
100	4
101	5
110	6
111	7
1000	8

Traditionally, we **read binary numbers left to right**, as opposed to right to left in our traditional decimal system.

Now you might notice the following trend:

Binary	Decimal
0	0
10	2
100	4
1000	8
10000	16
100000	32
1000000	64
…	

Every time we add a 0 to the right-hand side of the one, the number gets multiplied by 2. Mathematically the formula would then look like the following:

$$2^n$$

Where n is the number of 0's you add to the right of the number. So, let's look at this same table except with the formula filled in on a new column:

Binary	Decimal	2^n	Expanded
0	0	0	0
10	2	2^1	2
100	4	2^2	$2 * 2$
1000	8	2^3	$2 * 2 * 2$
10000	16	2^4	$2 * 2 * 2 * 2$
100000	32	2^5	$2 * 2 * 2 * 2 * 2$
1000000	64	2^6	$2 * 2 * 2 * 2 * 2 * 2$
…			

But that only gets part of the spectrum, so how do we represent the other numbers? By continuing the trend in our first chart.

But can we represent that trend mathematically? Yes, we can. The formula is the following and will give our value in decimal:

$$(x_n * 2^n) + (x_{n-1} * 2^{n-1}) + \cdots + (x_1 * 2^1) + (x_0 * 2^0)$$

Example:

Let's try an example the binary number 110111_2 to decimal:

1. We can see there are 6 digits so our initial formula will look like the following:
   $(x_5 * 2^5) + (x_4 * 2^4) + (x_3 * 2^3) + (x_2 * 2^2) + (x_1 * 2^1) + (x_0 * 2^0)$
2. Let's simplify it:
   $(x_5 * 32) + (x_4 * 16) + (x_3 * 8) + (x_2 * 4) + (x_1 * 2) + (x_0 * 1)$
3. Now what is x_5 and the others? Why the values at that position. Let's fill them in:
   $(1 * 32) + (1 * 16) + (0 * 8) + (1 * 4) + (1 * 2) + (1 * 1)$
4. And simplify:
   $32 + 16 + 4 + 2 + 1 = 55_{10}$

And that is our answer when we convert the binary number 110111_2 to decimal. While this isn't critical, because you can just plug this into a calculator it is nice to know for when you don't have access to one.

Decimal to Binary:

But what if your asked to go from decimal to binary? Do you remember how to divide things? Here are the basic steps. We will do an example:

1. Divide the number by 2
2. Get the integer quotient for the next iteration.
3. Get the remainder for the binary digit.
4. Repeat steps 1-3 until the quotient is 0.

Example Decimal to Binary:

Convert 13_{10} to binary:

Divide by 2	Quotient	Remainder	Bit #
13/2	6	1	0
6/2	3	0	1
3/2	1	1	2
1/2	0	1	3

Answer: 1011_2

Left Shift and Right Shift:

To look at this table again:

Binary	Decimal	2^n	Expanded
0	0	0	0
10	2	2^1	2
100	4	2^2	$2 * 2$
1000	8	2^3	$2 * 2 * 2$
10000	16	2^4	$2 * 2 * 2 * 2$
100000	32	2^5	$2 * 2 * 2 * 2 * 2$
1000000	64	2^6	$2 * 2 * 2 * 2 * 2 * 2$
…			

Whenever you add a 0 to the right-hand side you multiply the number by 2 and conversely when you remove a zero you divide the number by 2. These are special operations called "shifting" and is unique only to binary numbers because you are "shifting" the numbers to the left or right.

Adding a zero is left shift (multiply by 2).

Removing a zero is a right shift (divide by 2)

The syntax for them in programming is:

$$a \ll b$$
$$a \gg b$$

Where a is the original number and b is how many zeros you are adding on or removing.

Since, our brains don't work with binary we will use decimal numbers where appropriate.

Examples:

Left Shift:

$$3 \ll 2$$

This means we want to multiply the binary representation of 3 by adding 2 zeroes to the right of it:

$$3_{10} = 11_2$$

And adding two zeroes is:

$$1100_2$$

And then to convert back to decimal:

$$1100_2 = 12$$

Right Shift:

$$12 \gg 2$$

Similar example as the above but we just take off two zeroes instead of adding two zeroes from the right:

$$1100_2 = 12$$
$$11_2 = 3\ (removing\ two\ zeroes)$$

Alternative Method for Left and Right Shift:

The alternative way was to realize that if we are multiplying/dividing by 2 depending on the number on the right hand side of the equation then we can simplify our equation to the following rather than converting between binary (but the above is the underlying method):

$$a \ll b \ (generalized\ form\ of\ left\ shift)$$
$$a \ll b == a * 2^b$$

Or

$$a \gg b \ (generalized\ form\ of\ right\ shift)$$
$$a \gg b == \frac{a}{2^b}$$

Logical Operators:

You've probably seen these in Python:

and, or, not, xor

And maybe wrote a few programs that required them for such things as:

if a and b:
if a or b:
while a or b:
for a in range(10):

Or the like. Now those things return either a true or false and depending on whether it's true or not it'll do something different. Binary numbers do the same thing.

How? We will say the following about the digits in binary:

Binary Digit	Value
0	False
1	True

You can try this in Python for the logical statements and the above table will be true, although Python is a little bit different when you try other values:

"If the number is not 0 then its true. Otherwise its false".

With that out of the way lets go onto the logical operators themselves. I will introduce a sentence then explain it.

The sentences will be divided into two parts: **the left hand side (LHS), the "operator", and the right hand side (RHS)**

And:

"The sky is blue **(LHS)** and **(operator)** the grass is green **(RHS)**"

We can say that the two are very good assumptions about the sky and grass, and therefore the statement is overall true. However, let's say we said the following instead:

"The sky is green and the grass is green"

We can see that this sentence is blatantly false. So, let's generalize it:

LHS	RHS	And (Truthfulness)
False	False	False
False	True	False
True	False	False
True	True	True

As we can see the sentence is true if and only if both the left-hand side and right-hand side of the sentence is true.

Or:

"The sky is blue (LHS) or (Operator) the grass is green (RHS)"

Like above, these are good assumptions to be making and both of them happen to be true so the overall "truthfulness" of the sentence is correct. However, let's change it slightly:

"The sky is green or the grass is green"

The LHS is definitely false but the right-hand side is definitely true. However, by the definition of "or", the overall sentence is still true. So when is it false? Something like the following:

"The sky is green or the grass is blue"

Both sides are false so the overall sentence is wrong. So let's generalize it like usual:

LHS	RHS	Or (Truthfulness)
False	False	False
False	True	True
True	False	True
True	True	True

As we can see, the sentence is overall truthful if at least one of the sides of the sentence is true.

Not:

"The sky is blue"

What not does is quite simple, it negates the sentence and makes a true statement false and false statement true:

"The sky is **not** blue"

And the table is rather simple as well:

Sentence	Not
True	False
False	True

XOR:

This one is almost strictly a binary number only one. Basically it says that only one side can be true for the sentence to be true, otherwise the sentence is false:

LHS	RHS	XOR (Truthfulness)
False	False	False
False	True	True
True	False	True
True	True	False

Logical Operators and Relationship to Binary:

So, remember this table and how we decided the "truthfulness" of a binary digit?

Binary Digit	Value
0	False
1	True

We can apply this same exact table to the logical operators that we introduced above by replacing all False to 0 and all True to 1. So let's give an example:

$$1101\ AND\ 1001$$

If we were to write out like the LHS and RHS:

Digit # (R->L)	LHS Digit Value	RHS Digit Value	AND
0	1	1	1
1	0	0	0
2	1	0	0
3	1	1	1

Following the chart above under AND by replacing True with 1 and False with 0 our new number is then:

$$1001_2$$

So, whenever you AND, XOR, OR, NOT any two binary numbers together you apply the operator to each digit individually and that will be your new number.

Adding and Subtracting in Binary:

$$110_2 + 111_2 \text{ and } 1110_2 - 111_2$$

Adding and subtracting is exactly the same thing as if it was in the decimal system. Except when the total value hits 2 (instead of 10), you carry the one over or borrow the one.

For adding:

$$\begin{aligned} &110 \\ &+111 \\ &= 1101 \end{aligned}$$

For subtracting:

$$\begin{aligned} &1110 \\ &-111 \\ &= 111 \end{aligned}$$

Multiplying Binary:

$$110_2 * 111_2$$

Multiplying is the same thing as in the decimal system as well. Just add zeroes as you move right to left then add everything at the end (zeroes were added to line everything up nicely):

$$\begin{array}{r} 110_2 \\ * \quad 111_2 \\ \hline 110_2 \\ + \quad 1100_2 \\ + 110000_2 \\ \hline 101010_2 \end{array}$$

Dividing Binary:

The author has elected to not bother showing this part as he "considered" it not very useful and will just add length to it.

Chapter 3: Hexadecimal System

Introduction

You've looked at a binary system (base-2), and now you will look at a hexadecimal system (base-16). The difference? You get more digits to play with and a more compact number system (read fewer digits to represent the same number as you would in binary or our decimal system). And these are your digits:

$$0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A\ (10), B(11), C(12), D(13), E(14), F(15)$$

The number in brackets is their equivalent decimal number. Like in binary here is a chart and their equivalent values in decimal:

Hexadecimal	Decimal	Hexadecimal	Decimal
0	0	14	20
1	1	15	21
2	2	16	22
3	3	17	23
4	4	18	24
5	5	19	25
6	6	1A	26
7	7	1B	27
8	8	1C	28
9	9	1D	29
A	10	1E	30
B	11	1F	31
C	12	20	32
D	13	21	33
E	14	22	34
F	15	23	35
10	16	24	36
11	17	25	37
12	18	26	38
13	19	…	…

While, it's not immediately clear, due to how few numbers shown in this chart, but you'll eventually see that hexadecimal is a couple digits shorter than decimal at high enough numbers.

Adding, Subtracting and Multiplying:

The concepts are the same as the above in the previous chapter.

Logical Operators in Hexadecimal:

Remember our factoid in chapter 1? *"All numbers are representable in any other base*". Therefore, if we took a number that's in hexadecimal and converted it to binary then we could apply the logical operators as above in our chapter on binary (which is exactly what you should be doing).

Relationship to Binary:

Hexadecimal wasn't chosen randomly as it turns out. Let's look at 4 binary digits with all 1's:

$$1111$$

What is the value of this in decimal? 15.

Coincidence? I think not.

What is it in hexadecimal? **F**.

Huh… looks awfully familiar. Let's look at a larger number. How about 8 digits (separated by spaces into 4 binary digit groups):

$$1111\ 1111$$

What is the value of this? 255.

What about in hexadecimal? **FF**.

Let's generalize it:

"Every 4 binary digit represents 1 hexadecimal digit."

Conversely the following is also true then:

“Every hexadecimal digit is 4 binary digits.”

But what if you have an odd number like this in binary (again separated into groups of 4 for clarity):

11 1111

Well, we can argue that this is equivalent:

0011 1111

to the above and thus the same rule still applies.

Chapter 4 Summary and Relationship to Programming:

Summary:

You've just looked at binary and hexadecimal and gotten a relatively tiny taste of both namely how to add them, multiply them, basic math operations that we can do in decimal system. From here you can do the assignments that are in the other PDF (do not cheat and use an online calculator because that won't help cement the problems and the author chose very nice numbers that shouldn't take more than 5 minutes each to do).

Relationship to Programming:

As you will learn in the next project, binary is how the computers view all numbers and is unable to differentiate a "character" from a "number" from a "string" from a "Boolean" or even a positive and negative number. Those things are just for you, the programmer's, convenience.

The thing to realize is that the computer can't use all the numbers possible (unless your programming in Python, but that's more like a higher upper limit). It is limited in the number of total binary digits it can have (either 32 binary digits or 64 binary digits). If you were paying attention than the maximum number it can represent is, assuming 32 binary digits:

$$2^{31} + 2^{30} + 2^{29} + \cdots + 2^{0} = 4294967295_{10}$$

Or in hexadecimal:

$$FFFFFFFF$$

But, how do we represent a negative number? That's for the next project.

Appendix A: Formal(ish) Proof of Two Facts at the Beginning

Suppose you are given a positive integer, **x**. It doesn't matter how its written, as long as you can identify it (meaning that the original number system it came from doesn't matter). Now you want to convert it to any other base then it is equivalent to the following equation:

$$x = \sum_{i=0}^{n} b_i B^i$$

Where **{b_i}** are the digits of **x** for the Base, **B** > 1.

Now notice that if you were to divide the following by B, then by Euclidean division (for some integer a and b != 0 there exists a q and r such that $a = bq + r$):

$$x = x^0$$
$$\frac{x}{B} = \frac{x^0}{B}$$
$$x^0 = r_0 + B_x^{(1)}$$

For some integer $x^{(1)} < x^{(0)}$ and $r_0 < B$. Now if we were to divide x^1 by B recursively you end up with the following equation:

$$x = r_0 + B\left(r_1 + B\left(r_2 + B\left(\ldots + B(r_n)\right)\right)\right)$$

Where $0 \leq r_i < B$. Therefore by virtue of our first equation it necessarily means that: $b_i = r_i$.

The above proves fact #2 and by virtue of being able to represent any number in any base then we can convert any base to any other base with a slight extension to the above.