Alliants’ FizzBuzz challenge

Here's a roundup of FizzBuzz and some of the things that I found with the exercise. And a small writeup of what you saw (for non-programmers & programmers!).

Assignment
The first thing non-programmers should've seen are variables, which are names for things stored in memory (RAM). Variables in ruby are lowercase words. For example:

```
number = 10
```

This assigns the word number to the value ten, and places 10 within memory. Next time you want to access that memory location, you can just use the variable number. If another part of the program changes the variable - by resigning it, calling `number` again in the program will give us the new number.

Testing for Fizz, Buzz or both
The core of FizzBuzz is the part which tests a number to see whether it is
The most common approach was an **if/else ladder.** This is where a if statement is followed by successive else conditions. If none were met, the number was printed at the end.

```
number = 10

if number % 3 == 0 and number % 5 == 0

 puts "FizzBuzz"

elsif number % 3 == 0

 puts "Fizz"

elsif number % 5 == 0

 puts "Buzz"

else

 puts number

end
```

a few people, instead of directly declaring the outcome noticed you can start with a blank string, and then add Fizz then Buzz to it if it meets the % 3 == 0 and % 5 ==0 conditions, eliminating the need to test for both.

```
number = 10
result = ""
if number % 3 == 0
 result += "Fizz"
end
if number % 5 == 0
```

```
 result += "Fizz"
end
```

Ruby is very flexible in its semantics, using the principle of least surprise (basically - if it makes logical sense in the structure - it will work in Ruby). This is condensable to three lines. Chris had a similar solution.

```
number = 10
result = ""
result += "Fizz" if number % 3 == 0
result += "Fizz" if number % 5 == 0
```

However, by eliminating the if/else latter we now need to test if the variable result is empty to see whether we use the number.

```
if result = ""
 result = number.to_s
end
```

This tests if *result* is an empty string (of letters) and if so, it uses the number as the result. However the number is a different data type (a number). For non-programmers this may sound weird. But 10 the number is different from 10 the word (a one followed by a zero). Data types tell the programming language what can be done with the data. Numbers are good for arithmetic, strings for holding values like names, addresses and results like FizzBuzz. to_s is a ruby command to convert the number to a string.

Loops and Iterators
There are many ways to iterate through a snippet of code. The most common in all languages are the for and while loops. There are many types of ways to loop through code. You can use an if statement to create a loop. But there are cleverer ways. A traditional way is to use a conditional loop; this is where the program keeps on looping until a condition is met (*until* loop), or stops being met (*while* loop). An unconditional loop just keeps on looping until the program is forced to stop or crashes!

```
while number < 101
 # do the test here!
 # then increase the number by 1!
 number = number + 1
end
```

This will keep on looping until the number exceeds 100. A conditional loop is best known when there is a known stop condition and the number of iterations is not known. Although this will work, better would be another type of loop called the for loop. The *for* loop is different to a conditional loop in that is has an explicit loop counter. This is used when the number of times you want to loop is known. In a lot of scenarios the *for* and *while* loops can be used interchangeably.

Ruby has lots of cool ways of declaring for loops:

The classic for loop:

```
for number in 1..100
 # do the test here!
 # then increase the number by 1!
end
```

Upto for loop

```
1.upto(100) do |number|
 # do the test here!
 # then increase the number by 1!
end
```

Times iterator

```
100.times do |number|
 # do the test here!
 # then increase the number by 1!
end
```

Each of these uses the values [1, 2, 3, 4, ... 100] explicitly.

Those more experienced in Ruby tended to use block iterator methods rather than looping constructs. These iterate over a list of values. The most simple is the each method, which iterates over a block (so no incrementer is required).

```
(1..100).each do |number|
  # do the test here!
end
```

One result!
So with that in mind, here's an example of the FizzBuzz code. Notice the test (to see if the number is empty) is reduced using the helper method .empty?

```
(1..100).each do |number|
 result = ""
 result += "Fizz" if number % 3 == 0
 result += "Buzz" if number % 5 == 0
 puts ( result.empty? ? number : result)
end
```

Ed’s challenge!
Ed pointed out that in *his* day, players had to say *roman numerals* instead of the numbers; unless it was divisible by three (Fizz) or ***seven*** (Buzz) or both (FizzBuzz). In Ed's day numbers hadn't been invented, nor had computers - so this was some challenge. To solve this, I reused the program above (changing 5 to a 7) and simply introducing a method to do a conversion from numeric data type to a roman numeral string.

Rather than show how this method works - it is the subject of the next test!!

The result:

```
I
II
Fizz
IV
V
Fizz
Buzz
VIII
Fizz
```

```
X
XI
Fizz
XIII
Buzz
Fizz
XVI
XVII
Fizz
XIX
XX
FizzBuzz
XXII
XXIII
Fizz
XXV
XXVI
Fizz
Buzz
XXIX
Fizz
XXXI
XXXII
Fizz
XXXIV
Buzz
Fizz
XXXVII
XXXVIII
Fizz
XL
XLI
FizzBuzz
XLIII
XLIV
Fizz
XLVI
XLVII
Fizz
Buzz
L
Fizz
LII
LIII
Fizz
LV
Buzz
Fizz
LVIII
LIX
Fizz
LXI
LXII
FizzBuzz
LXIV
LXV
Fizz
LXVII
LXVIII
Fizz
```

```
Buzz
LXXI
Fizz
LXXIII
LXXIV
Fizz
LXXVI
Buzz
Fizz
LXXIX
LXXX
Fizz
LXXXII
LXXXIII
FizzBuzz
LXXXV
LXXXVI
Fizz
LXXXVIII
LXXXIX
Fizz
Buzz
XCII
Fizz
XCIV
XCV
Fizz
XCVII
Buzz
Fizz
C
```

Thanks all!