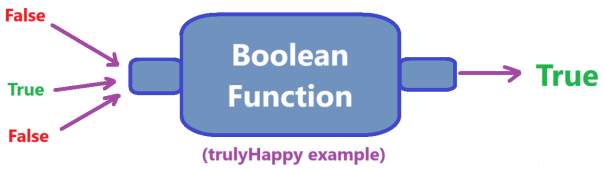
Logic with Numbers and Strings

1 Breaking Problems Into Cases

We've been working on functions that take in `True`s and `False`s and spit out `True` or `False`.



`trulyHappy` was a problem from the previous homework.

In these problems, because we only had `True` and `False` as inputs, the amount of possibilities we had to handle was small. But what if we took in numbers as inputs? Then there would be infinite cases to handle for every possibility!

```
1  def function(num):
2      if num == 0:
3          (Code)
4      if num == 1:
5          (Code)
6      if num == 2:
7          (Code)
8
9      ...And so on!!
```

For example, what if we wanted to write a function where if the input number was even, we return twice the number, and if the number is odd, we return thrice the number. It might be tempting to write:

```
1       def coolFunction(num):
2           if num == 0:
3               return 0        (2 · 0)
4           if num == 1:
5               return 3        (3 · 1)
6           if num == 2:
7               return 4        (2 · 2)
8           if num == 3
9               return 9        (3 · 3)
10          if num == 4
11              return 8        (2 · 4)
12          if num == 5
13              return 15       (3 · 5)
14
15
16          ... And so on!
```

But this would be infinite code to have to write. Instead, what if we just wrote code to handle if the number is odd and code to handle if the number is even? Reminder that the % operator in Python gets the remainder. We can think of what we have to do in the following chart:

Input	If Statement	Output
`num` is even	`if num % 2 == 0`	`2 * num`
`num` is odd	`if num % 2 == 1`	`3 * num`

Therefore, our code would be:

```
1       def coolFunction(num):
2           if num % 2 == 0:
3               return 2 * num
4           elif num % 2 == 1:
5               return 3 * num
```

That's a lot better than infinite code to write! You can see the power of cases here. Via cases, we could handle every number possible, just like how we could handle every possible input of `True` and `False` in boolean problems.

2 Practice with a CodingBat Problem

Let's look at `Warmup-1` → `sum_double`:

> Given two int values, return their sum. Unless the two values are the same, then return double their sum.
> Example:
>
> `sum_double(1, 2)` → 3
> `sum_double(3, 2)` → 5
> `sum_double(2, 2)` → 8

If we handled every possible input, we would have ∞ numbers times ∞ numbers equals ∞ possibilities. But once again, we can divide into cases:

Input	If Statement	Output
`a` and `b` are different	`if a != b`	`a + b`
`a` and `b` are the same	`if a == b`	`2 * (a + b)`

In programming, to say $a \neq b$, we say `a != b`, which is what we have above. We can use this chart to write our solution:

```
1       def sum_double(a,b):
2           if a == b:
3               return 2 * (a + b)
4           elif a != b:
5               return a + b
```

Notice that we can simplify this code into:

```
1       def sum_double(a,b):
2           if a == b:
3               return 2 * (a + b)
4           else:
5               return a + b
```

Because `return` immeadetly ends a function, we don't really need the `else` here first, meaning the code can be further simplified into:

```
1       def sum_double(a,b):
2           if a == b:
3               return 2 * (a + b)
4           return a + b
```

Here, we handle the exception first (the numbers are exactly the same), and then we handle the main case (where the numbers are different, which is the majority of the time) afterwards.

3 Sometimes We Don't Need Cases

Consider this problem:

> Write a function `explode` that takes in two numbers a and b and returns the sum of a^b and b^a.

There's only one scenario here: no matter what the inputs are, we always return $a^b + b^a$. So our code isn't:

```
1       def explode(a,b):
2           if (a == 0 and b == 0):
3               return 0**0 + 0**0
4           if (a == 0 and b == 1):
```

```
5               return 0**1 + 1**0
6           if (a == 1 and b == 0):
7               return 1**0 + 0**1
8
9           ...and so on.
```

Instead, our code is:

```
1       def explode(a,b):
2           return a**b + b**a      (This handles every scenario!!)
```

Try to break a problem down into the main scenarios. Is there an equation that relates the input to the answer? Or do you need multiple equations depending on the input, requiring cases?

4 Logic with Strings

Just like how we could handle problems with numbers as inputs, we can also handle problems with strings (English words) as inputs. Despite how there's infinite possibilities of input—as there's infinite English sentences you could give as input—we can still break a problem into cases (or potentially solve it without cases). Consider CodingBat problem `String-1` → `make_abba`:

> Given two strings, a and b, return the result of putting them together in the order abba, e.g. `"Hi" and "Bye"` returns `"HiByeByeHi"`.
>
> ```
> make_abba('Hi', 'Bye')   → 'HiByeByeHi'
> make_abba('Yo', 'Alice') → 'YoAliceAliceYo'
> make_abba('What', 'Up')  → 'WhatUpUpWhat'
> ```

First, notice that in Python, we can use `'` and `"` interchangably. This isn't the case in other langauages, but for Python, it's totally okay to say `"cat"` or `'cat'`: both are strings and are the same thing to the computer.

Let a and b represent the input strings. No matter what a and b are, we always return $a + b + b + a$. So our code shouldn't be:

```
1       def make_abba(a,b):
2           if a == "Hi" and b == "Bye":
3               return "HiByeByeHi"
4           if a == "Yo" and b == "Alice":
5               return "YoAliceAliceYo"
6           if a == "What" and b == "Up:
7               return "WhatUpUpWhat"
8
9           ...and so on forever.
```

Instead, our code is:

```
1       def make_abba(a,b):
2           return a + b + b + a
```

and this code will handle every scenario! Trust that the algebra will work for any input. Your computer is smart. It can do it even if it looks like variables to you.

5 Tools to Help Us with Numbers

For comparing numbers and writing if-statements, there's a lot of useful operators:

Operator	Math Meaning
`a == b`	$a = b$
`a != b`	$a \neq b$
`a < b`	$a < b$
`a <= b`	$a \leq b$
`a > b`	$a > b$
`a >= b`	$a \geq b$

These operators all give an output of `True` whenever the math equation is correct. So `4 < 5` would be `True`, so the following if-statement

```
1       a = 4
2       b = 5
3       if a < b:
4           (Code)
```

The code inside the `if` would indeed happen. Similarly, `3 >= 3` as $3 = 3$, so the code

```
1       c = 3
2       d = 3
3       if c >= d:
4           (Code)
```

would also have its code run.

6 Tools to Helps Us with Strings

Just like with numbers, we can use `== and !=` to compare strings. But there's other operations as well. Consider the following string:

```
s = "a racecar"
```

Including the spaces, there are 9 characters (called `chars` in programing) in this string:

A line is put under the space in this diagram for counting purposes.

To get a letter from a string `s`, you write `s[i]`, replacing i with the number of the letter you want. However, if you actually tried `s[3]`, you would get `'a'` and not `'r'` as expected. This is because in programming, we count strings starting with the first letter as index 0. We start at 0 because it was easier to build early computers this way. So here is a diagram of the letters correctly numbered:

Notice that `s[2]` is `'r'`, not `s[3]`.

To get a section from a larger string, we do `s[i:j]`. So for example, `s[2:2+4]` would be the string `"race"`. This is called a *substring*.

I wrote $2 + 4$ because there's four letters in race. Since $2 + 4 = 6$, we could have also written `s[2:6]`. But notice that `s[6] == 'c'`, not `'e'`! So when we find substrings, we give the index of the first letter and then the index of the letter *one after* we want to stop at. So as another example:

```
s[6:9] == "car"
```

To get the length of a string, we use `len()`. So here, `len(s) == 9`. This means we could also have written:

```
s[6:len(s)] == "car"
```

To summarize,

Code	Meaning
`a == b`	Are `a` and `b` the same words.
`a != b`	Are `a` and `b` different words.
`s[i]`	The ith $+ 1$ letter.
`s[i:j]`	The substring starting at index i and ending one before index j.
`len(s)`	The length of the string `s`.

There's a lot more we can do with strings, but we'll learn about them in a future lesson.

Let's use what we learned to solve a problem. Consider the following CodingBat problem, which is `String-1` → `without_end`

> Given a string, return a version without the first and last char, so `"Hello"` yields `"ell"`. The string length will be at least 2.
>
> ```
> without_end('Hello') → 'ell'
> without_end('java') → 'av'
> without_end('coding') → 'odin'
> ```

You might be tempted to handle every possibility:

```
1     def without_end(s):
2         if s == "Hello":
3             return "ell
4         if s == "python":
5             return "ytho"
6         if s == "cute cat":
7             return "ute ca"
8
9         ...And so on forever.
```

But that would be infinite code to write to handle every possible English string of words! Instead, we can recognize there's only **one case**: remove the first and last letters, which we can do with substrings.

We want to skip the first letter, which is letter 0, meaning we should start at 1. And we want to skip the last letter. The last letter of any string is `len(s) - 1`: For example,

```
1       def without_end(s):
2           return s[1:len(s)-1]
```

7 Homework

It's time for CodingBat! Here's what I would like you to do:

Numbers:

1. `Warmup-1` → `parrot_trouble`
2. `Warmup-1` → `diff21`
3. (A step up) `Warmup-1` → `makes10`
4. (Challenging) `Warmup-1` → `near_hundred`

Strings:

1. `String-1` → `hello_name`
2. `String-1` → `extra_end`
3. `String-1` → `first_two`
4. `String-1` → `first_half`
5. (A step up) `String-1` → `make_out_word`

6. (Similar difficulty) `String-1` → `make_tags`