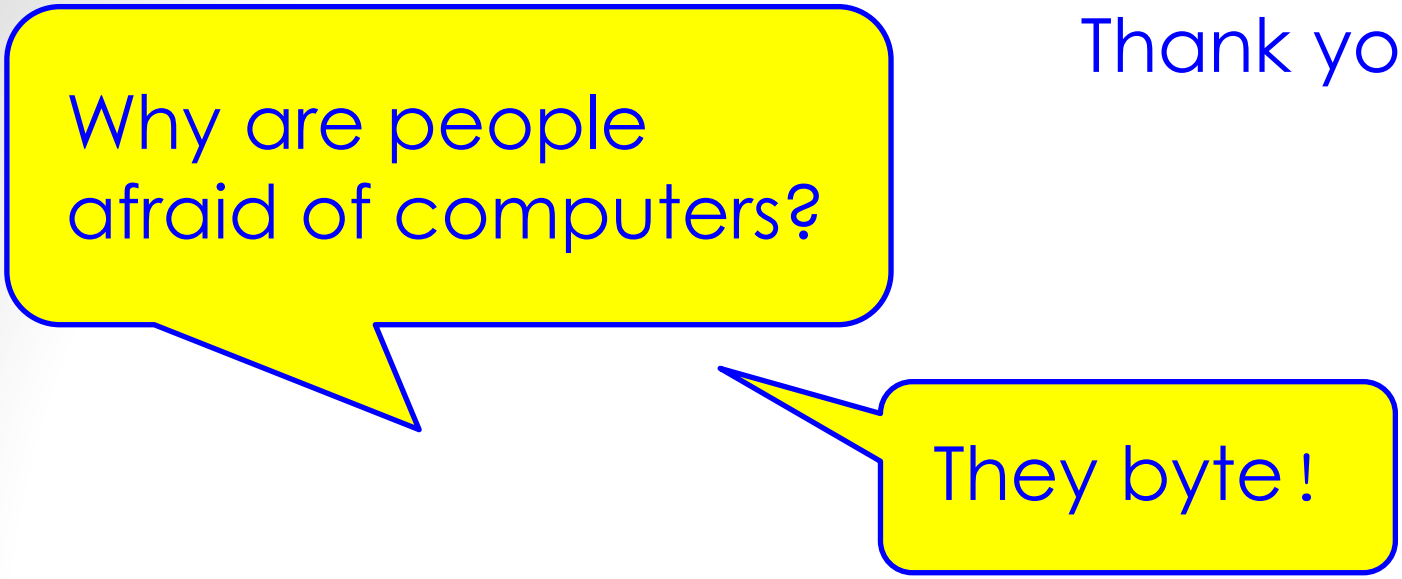
Thank you Luke!




Source: https://scarymommy.com/computer-jokes, joke #31

# CMPT 120

Lecture 9 – Cryptography and Encryption –
The realm of secret codes
Python – Arithmetic operators, order of evaluation (operator precedence) and string/list manipulation

# Last Lecture
# Improving our guessing game

- Wouldn't it be nice to play our guessing game many times without having to press **Run** over and over again?

- **New Problem Statement**
  - Write a guessing game, which allows a player to guess a number between 1 and 10 **in 3 guesses!**

- **Let's get coding!**

# Review - `range( )` function

- Very useful in `for` loop
- Syntax: `range([start,] stop [,step]))` (included / excluded)
- Produces a list of integers
- How does it work?
  - If **`theLength = 5`**
  - Then **`range(theLength)`**
    produces the following sequence: **`0,1,2,3,4`**

# Repeated code -> Bad idea!

If you need to repeat some of your code in your program, do not repeat the code itself by copying and pasting it, use a **loop** instead!

- What do you mean by ***repeated code***?
- If the **problem statement** is: List some movies, then Solution 1 would solve the problem using repeated code -> bad idea!

This semester, let's not do this! ☹

  - Solution 1:

```
print("Superman")
print("Frozen")
print("X-Men")
```

- Solution 2 would not -> good idea!

This semester, let's do this! ☺

  - Solution 2:

```
movies = ["Superman", "Frozen", "X-Men"]
for movie in movies:
  print(movie,"!")
```

# Todays' Menu

- Introducing **Cryptography and Encryption**
- Can we build programs that create secret(encrypted) messages using
  - Arithmetic operators
  - String and List indexing and slicing mechanism
  - etc…
- Let's see ☺

# Cryptography and Encryption - Secret codes

- When you log onto your SFU mail account, purchase goods on the Internet, check your grades or your bank account online, you are using cryptography.

- **Cryptography** is a field of study involving many disciplines such as mathematics, **computing science**, and engineering
- It encompasses various aspects such as data confidentiality, data integrity, and authentication

- **Encryption** is one of the aspects of cryptography that deals with the process of encoding a message using an algorithm

# Cryptography and Encryption - Secret codes

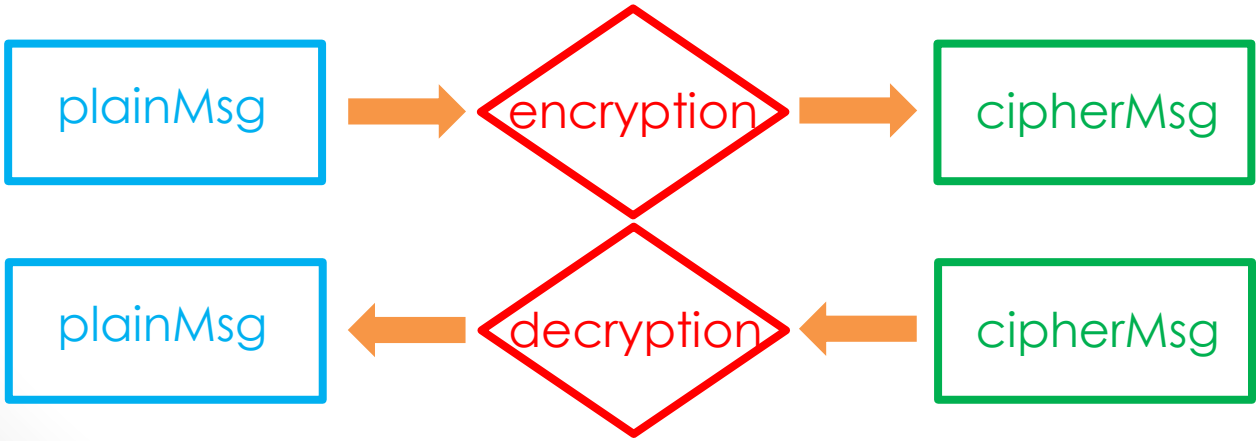
- In this unit, we'll see how we can **encrypt**/**decrypt** messages using Python programs

- To do so, we'll need to learn
  - How to manipulate characters in a string
  - How to manipulate elements in a list
  - How to calculate
  - How to loop in various ways
  - How to write our own **functions**

# Encryption - Secret codes

- Makes messages confidential, secret
- It does this by using an algorithm that transforms messages we can read into messages we can no longer read … and back to messages we can read

- We shall call
  - messages we can read **plainMsg**
  - messages we cannot read **cipherMsg**



# Let's give it a go!

- **Step 1 - Problem Statement:**
  - Write a Python program that **encrypts** messages using a **transposition algorithm** called **odd&even**

# Transposition algorithm odd&even

encryption

**Definition**: Algorithm that shuffles elements from their original positions in a sequence to new positions!

Transposition algorithm **odd&even**:

1. Get **plainMsg** from user
2. Create a **cipherMsg** that is made of 2 strings
   - **String1** contains the characters located in odd positions in **plainMsg**
   - **String2** contains the characters located in even positions in **plainMsg**
3. Lastly, concatenate these two strings:
   **cipherMsg** = **String1** + **String2**

# An example:

- Let's encrypt this message `"Hello, World!"`

# Let's give it a go! (cont'd)

- **Step 2 – Design**

  Transposition algorithm **odd&even**:

  1. Get **plainMsg** from user
  2. Create a **cipherMsg:**

     For each character in the **plainMsg**

     - If the character is at an odd position in **plainMsg**
       - Then this character goes into **String1**
       - Otherwise it goes into **String2**
  3. Lastly, concatenate these two strings: **cipherMsg** = **String1** + **String2**

# Let's give it a go! (cont'd)

- **Step 3 – Implementation**

  In order to implement our **encryption** algorithm, we need to know …

  1. String concatenation
  2. Arithmetic operator -> modulo operator
     - Example:

  3. Order of evaluation (precedence)
     - Example:

  4. Running count (**accumulator**) algorithm

# Let's give it a go! (cont'd)

- **Step 4 - Testing**

# Your turn:
# Let's decrypt our message!

- **Step 1 - Problem Statement:**
  - Write a program that **decrypts** messages that have been **encrypted** using the **transposition algorithm odd&even**

# Review - String **indexing**: positive indexing

How to access one string character at a time?

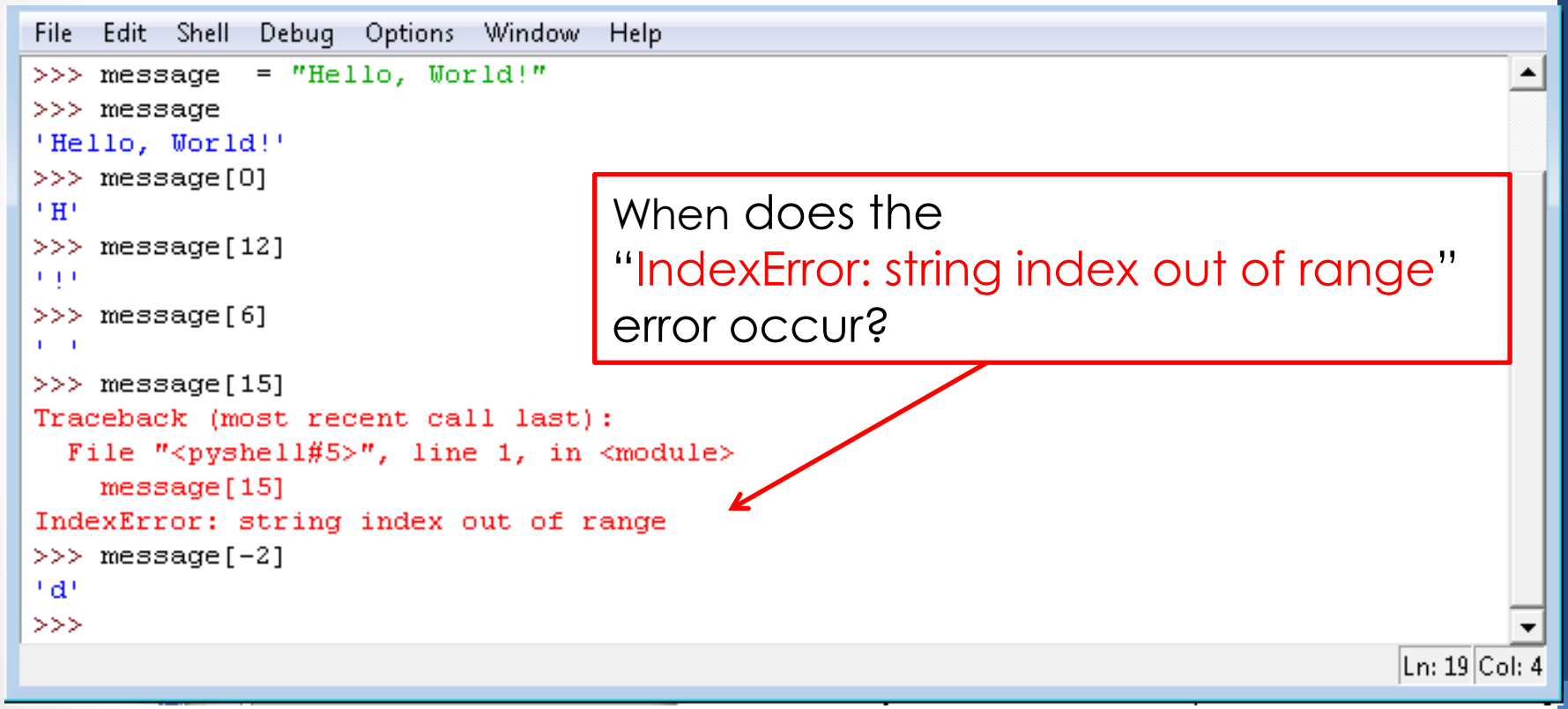
Answer: Use the index associated with the character as illustrated below:

positive indexing-> **index:** **0 1 2 3 4 5 6 7 8 9 10 11 12**

**Example:** `message = "Hello, World!"`

- So if we wish to access
  - The 1st character of the string, we use the index 0
  - The 2nd character of the string, we use the index 1
  - etc…

# Review - String **indexing**: positive indexing examples

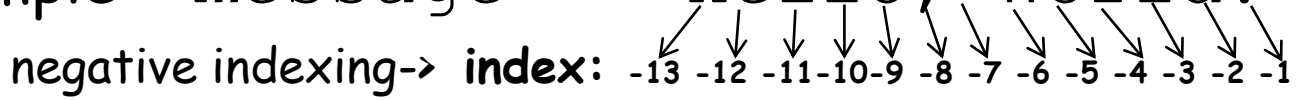



Careful: Positive index starts at **0**

# Review - String **indexing**: negative indexing

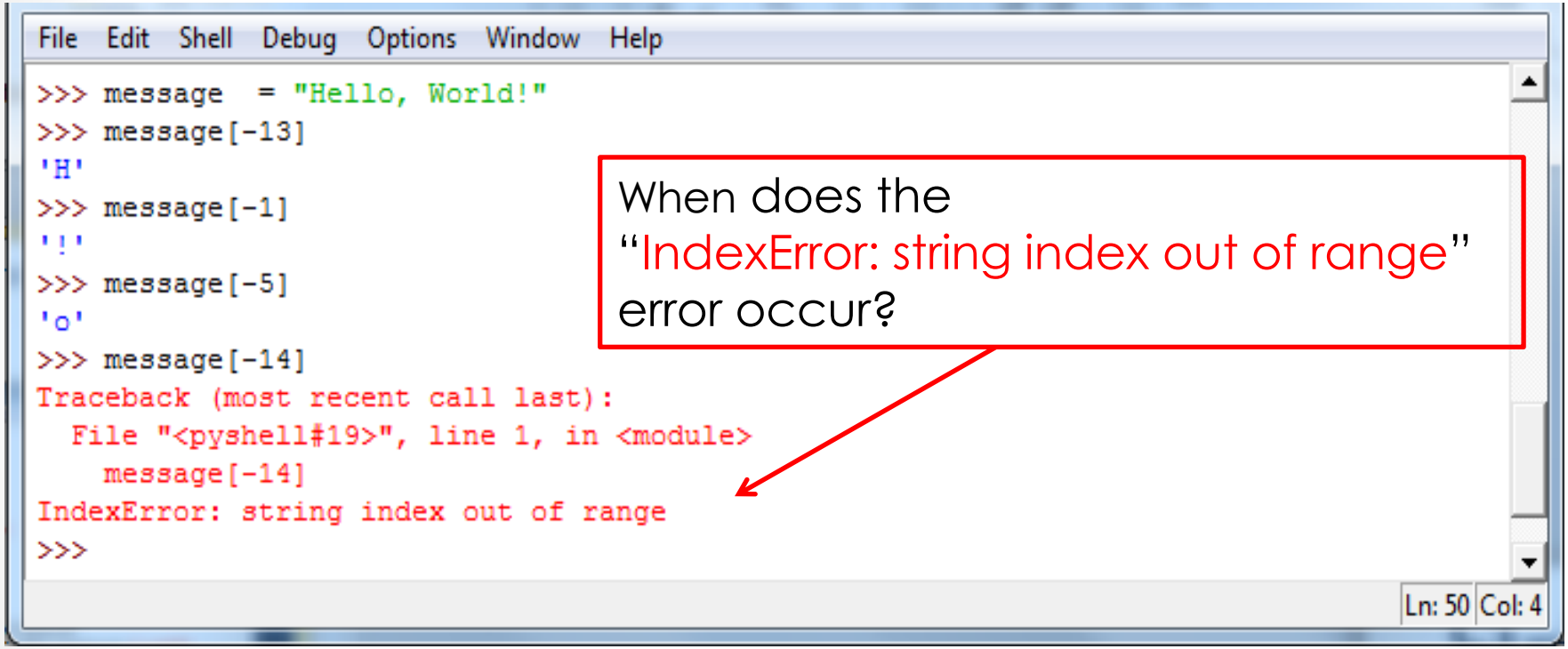
- There is another way we can use to access one string character at a time: negative indexing:

**Example:** `message = "Hello, World!"`




- So if we wish to access
  - The 1st character of the string, we use the index -13
  - The last character of the string, we use the index -1,
  - etc…

# Review - String **indexing**: negative indexing examples




Careful: Negative index starts at **-1**, **not 0**

# Review - String **slicing** (using positive indices)

How to access a section (slice) of a string at a time?

Answer: use indices to indicate the string slice

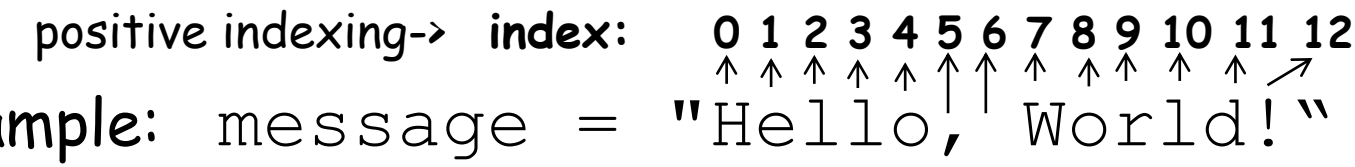
positive indexing-> **index:** **0 1 2 3 4 5 6 7 8 9 10 11 12**

**Example:** `message = "Hello, World!"`

**Syntax:** `<aString>[start : stop : step]`

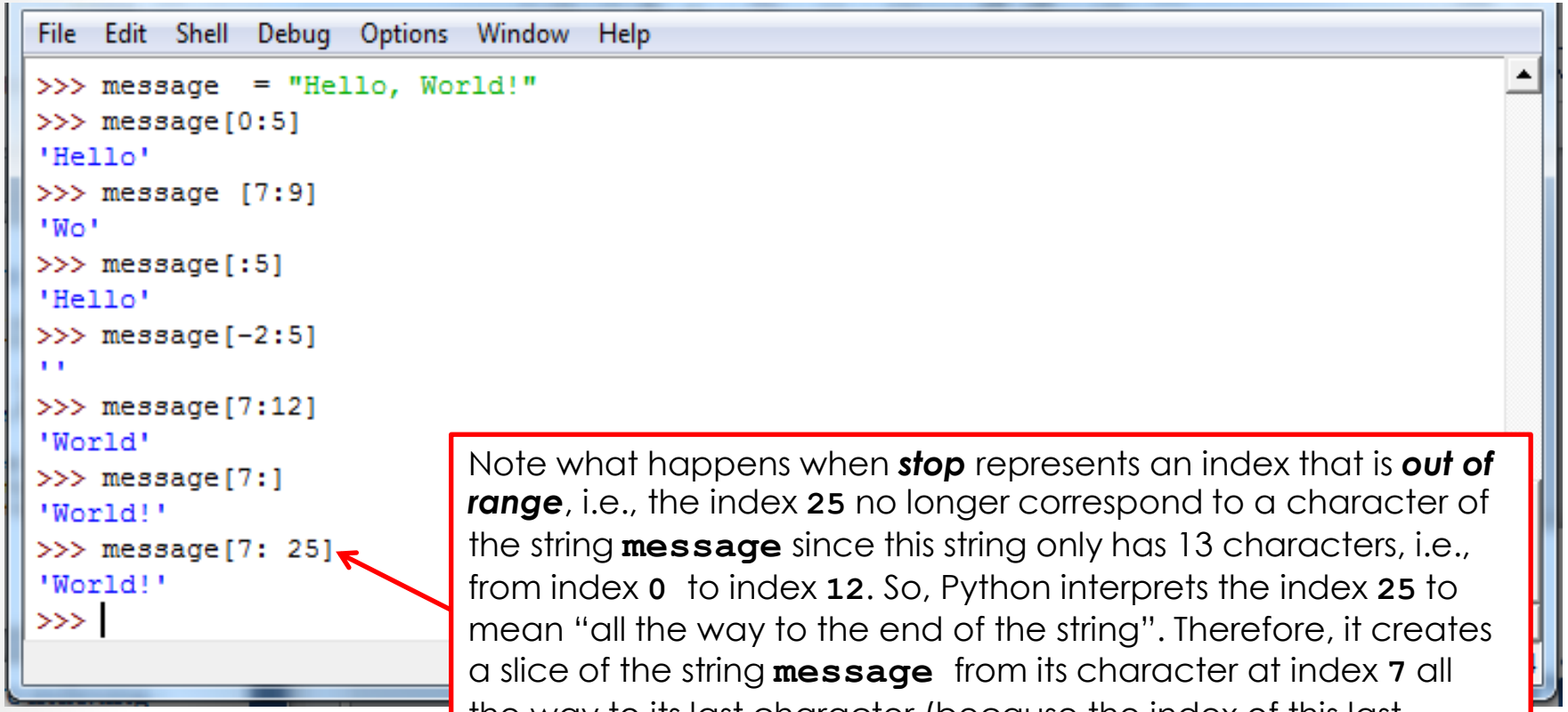
- `start`
- `stop`
- `step`
- Example: So if we wish to access the string slice "**`Hello`**", we use **`message[0:5]`**

# Review - How does String **slicing** works?




- **message[0:5]**
  - We use index **0** to indicate the ***start*** of the string slice
    - **Inclusive** -> the character at index **0** is included in the string slice
  - We use index **5** to indicate the stop of the string slice
    - **Non-inclusive** -> the character at index **5** is ***not*** included in the string slice

# Review - String **slicing** - Examples



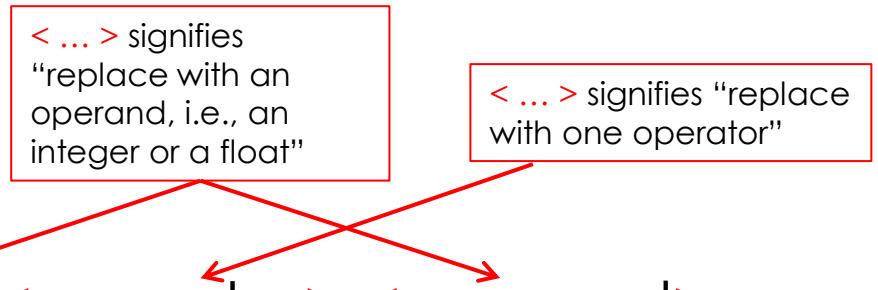

Note what happens when ***stop*** represents an index that is ***out of range***, i.e., the index **25** no longer correspond to a character of the string **message** since this string only has 13 characters, i.e., from index **0** to index **12**. So, Python interprets the index **25** to mean "all the way to the end of the string". Therefore, it creates a slice of the string **message** from its character at index **7** all the way to its last character (because the index of this last character is < **25**).

# Review - Strings (sequence) manipulation

| **Operation Name** | **Operator/ function** | **Comment** |
|---|---|---|
| concatenation | + | Combine strings together |
| repetition | * | Concatenate a string that is being repeated a number of times |
| indexing | [n] | Access an element of a string |
| slicing | [ : : ] | Extract a part of a string |
| length | *len(aString)* | Determine the number of characters in a string aString |

# Review: Arithmetic operators

- Addition: **+**
- Subtraction: **-**
- Multiplication: *****
- Division: **/**
- Floor division: **//**
- Modulus: **%**
- Exponentiation: ******




- **Syntax**: <operand> <operator> <operand>

- Running count (**accumulator**) algorithm
  - Example: `charCount = charCount + 1`
    `OR charCount += 1`

# Review: Order of Evaluation

**Highest precedence**

| | |
|---|---|
| (expressions...) | Parentheses **P** |
| x[index],<br>x[index : index],<br>x(arguments...) | Indexing (aka Subscription),<br>Slicing,<br>Call |
| ** | Exponentiation **E** |
| *, /, //, % | Multiplication, division (`float` and `int`), remainder **MDR** |
| +, - | Addition and subtraction **AS** |
| <, <=, >, >=, !=, == | Relational operators |
| not | Logical operator |
| and | Logical operator |
| or | Logical operator |

**Lowest precedence**

# Next Lecture

- Practice Exam #2 ☺