CMSC201
Computer Science I for Majors

Lecture 08 – Strings (and More)

Last Class We Covered

- Lists and what they are used for
  - Getting the length of a list
  - Operations like **`append()`** and **`remove()`**
  - Iterating over a list using a **`while`** loop

- Sentinel loops
- Priming read

Any Questions from Last Time?

Today’s Objectives

- To better understand the string data type
  - Learn how they are represented
  - Learn about and use some of their built-in functions
- To cover some other miscellaneous details
  - Learn about the importance of constants
  - Be able to implement **`while`** loops with Boolean flags

Strings

The String Data Type

- Text is represented in programs by the string data type
- A ***string*** is a sequence of characters enclosed within quotation marks (") or apostrophes (')
  - Sometimes called double quotes or single quotes

Getting Strings as Input

- Using **`input()`** automatically gets a string

```
>>> firstName = input("Please enter your name: ")
Please enter your name: Shakira
>>> type(firstName)
<class 'str'>
>>> print(firstName, firstName)
Shakira Shakira
```

Accessing Individual Characters

- We can access the individual characters in a string through ***indexing***
  - Characters are the letters, numbers, spaces, and symbols that make up a string

- The characters in a string are numbered starting from the left, beginning with 0
  - Just like in lists!

Syntax of Accessing Characters

- The general form is

  `strName[expression]`

- Where **`strName`** is the name of the string variable and **`expression`** determines which character is selected from the string

Quick Note: Python Interpreter

- Sometimes in class and the slides, you'll see use of Python's "interactive" interpreter
  - Evaluates each line of code as it's typed in

```
>>> print("Hello")
Hello
>>> 4 + 7
11
>>>
```

>>> is where the user types their code

lines without a ">>>" are Python's response

  - To use the interpreter, enable Python 3, then type "**python**" into the command line

Example String

0	1	2	3	4	5	6	7	8
H	e	l	l	o		B	o	b

```
>>> greet = "Hello Bob"
>>> greet[0]
'H'
>>> print(greet[0], greet[2], greet[4])
H l o
>>> x = 8
>>> print(greet[x - 2])
B
```

Example String

0	1	2	3	4	5	6	7	8
`H`	`e`	`l`	`l`	`o`		`B`	`o`	`b`

- In a string of **`n`** characters, the last character is at position **`n-1`** since we start counting with 0

- So how can we access the <u>last</u> letter, regardless of the string’s length?

```
greet[ len(greet) - 1 ]
```

Substrings and Slicing

Substrings

- Indexing only returns a <u>single</u> character from the entire string
- We can access a ***substring*** using a process called ***slicing***
  - Substring: a (sub)part of another string
  - Slicing: we are slicing off a portion of the string

Slicing Syntax

- The general form is

  `strName[start:end]`

- **`start`** and **`end`** must both be integers
  - The substring begins at index **`start`**
  - The substring ends **<u>before</u>** index **`end`**
    - The letter at index **`end`** is <u>not</u> included

Slicing Examples

0	1	2	3	4	5	6	7	8
H	e	l	l	o		B	o	b

```
>>> greet[0:2]
'He'
>>> greet[7:9]
'ob'
>>> greet[:5]
'Hello'
>>> greet[1:]
'ello Bob'
>>> greet[:]
'Hello Bob'
```

Specifics of Slicing

- If **`start`** or **`end`** are missing, then the start or the end of the string are used instead
- The index of **`end`** must come <u>after</u> the index of **`start`**
  - What would the substring **`greet[1:1]`** be?
    `''`
  - An empty string!

Forming New Strings - Concatenation

- We can put two or more strings together to form a longer string

- ***Concatenation*** "glues" two strings together

```
>>> "Peanut Butter" + "Jelly"
'Peanut ButterJelly'
>>> "Peanut Butter" + " & " + "Jelly"
'Peanut Butter & Jelly'
```

Rules of Concatenation

- Concatenation does <u>not</u> automatically include spaces between the strings

```
>>> "Smash" + "together"
'Smashtogether'
```

- Concatenation can <u>only</u> be done with strings!
  - So how would we concatenate an integer?

```
>>> "CMSC " + str(201)
'CMSC 201'
```

Uses for Concatenation

- **`input()`** only accepts a single string
  - Can't use commas like we do with **`print()`**
- In order to create a single string for **`input()`**, you must use concatenation

```
classNum = 201
grade = input("Grade in " + str(classNum) + "? ")
```

String Operators in Python

Operator	Meaning
`+`	Concatenation
`STRING[#]`	Indexing
`STRING[#:#]`	Slicing
`len(STRING)`	Length

- All of this also applies to lists!
  - Two lists can be concatenated together
  - A sublist can be sliced from another list

Just a Bit More on Strings

- Python has many, many ways to interact with strings, and we will cover them in detail soon
- For now, here are two very useful functions:

  `s.lower()` – copy of **`s`** in all lowercase letters

  `s.upper()` – copy of **`s`** in all uppercase letters

- Why would we need to use these?
  - Remember, Python is <u>case-sensitive</u>!

Constants

What are Constants?

- Constants are values that are **not** generated by the user or by the code
  – But are used a great deal in the program

- Constants should be ALL CAPS with a "**_**" (underscore) to separate the words
  – This follows CMSC 201 Coding Standards

Using Constants

- Calculating the total for a shopping order

```
MD_TAX    = 0.06
```

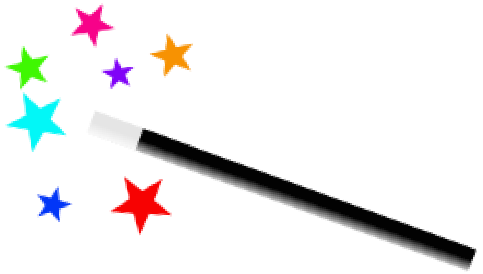
easy to update if tax rate changes

```
subtotal = input("Enter subtotal:")
subtotal = float(subtotal)
tax    = subtotal * MD_TAX
total = tax + subtotal
print("Your total is:", total)
```

we know exactly what this number is

“Magic” Numbers

- “Magic” numbers are numbers used directly in the code – should be replaced with constants



- Examples:
  - Mathematical numbers (pi, e, etc.)
  - Program properties (window size, min and max)
  - Important values (tax rate, maximum number of students, credits required to graduate, etc.)

"Magic" Numbers Example

- You're looking at the code for a virtual casino
  - You see the number 21
  - What does it mean?

```
if value < 21:
```
✗

- Blackjack? Drinking age? VIP room numbers?

```
if customerAge < DRINKING_AGE:
```
✓

- Constants make it easy to update values – why?
  - Don't have to figure out which "21"s to change

“Magic” Everything

- Can also have “magic” characters or strings
  - Use constants to prevent <u>any</u> “magic” values
- For example, a blackjack program that uses the strings “**H**” for hit, and “**S**” for stay

```
if userChoice == "H":
```
✗

```
if userChoice == HIT:
```
✓

  - Which of these options is easier to understand?
  - Which is easier to update if it’s needed?

Are Constants Really Constant?

- In some languages (like C, C++, and Java), you can create variables that CANNOT be changed
- This is <u>not possible</u> with Python variables
  - Part of why coding standards are so important
  - If you see code that changes the value of a variable called **`MAX_ENROLL`**, you know that's a constant, and *shouldn't* be changed

Where Do Constants Go?

- Constants go before **`main()`**, after your header comment

- All variables that aren't constants must be inside of **`main()`**

```
# File:    hw2_part1.py
# Author:  Dr. Gibson
# etc...

MAX  = 28
WEEK = 7

def main():
    date = int(input("Please enter day: "))

    if date >= 1 and date <= MAX:
        # etc...
main()
```

Boolean Flags

Complex Conditionals

- Sometimes, a **`while`** loop has many restrictions or requirements
  - Expressing them in one giant conditional is difficult, or maybe even impossible

- Instead, break the problem down into the separate parts, and use a single Boolean "flag" value as the loop variable

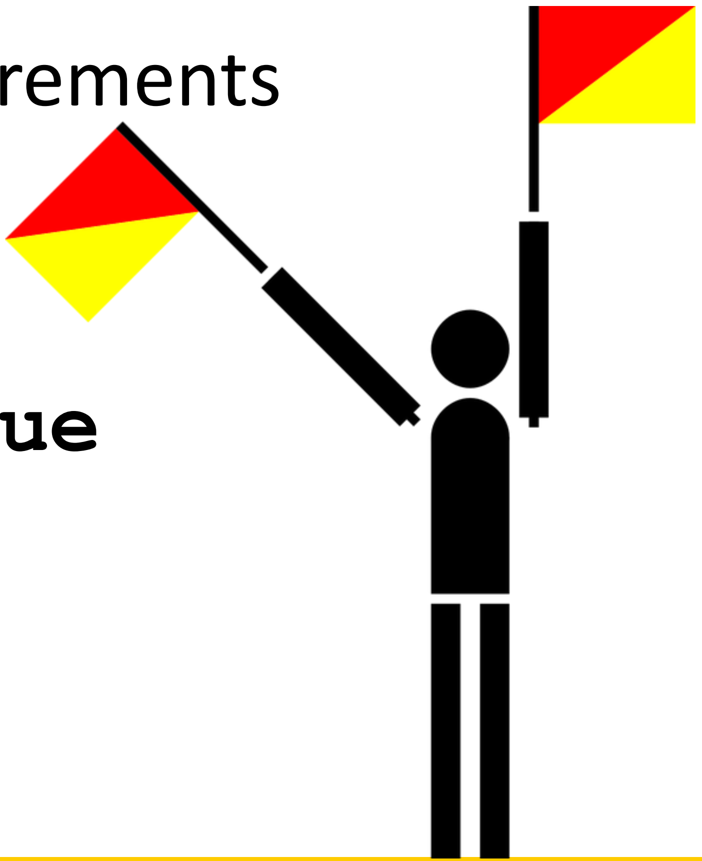
Complex Examples

- Multiple requirements to satisfy
  - Password must be at least 8 characters long, no longer than 20 characters, and have no spaces or underscores

- Multiple ways to satisfy the requirements
  - Grade must be between 0 and 100, unless extra credit is allowed, in which case it can be over 100

Boolean Flags

- A Boolean value used to control the while loop
  - Communicates if the requirements have been satisfied yet
- Value should evaluate to **`True`** while the requirements have <u>not</u> been met



General Layout – Multiple Reqs

- Start the **`while`** loop by
  - Getting the user's input
  - Assuming that all requirements are satisfied
    - (Set the Boolean flag so that the loop would exit)
- Check each requirement individually
  - For each requirement, if it isn't satisfied, change the Boolean flag so the loop repeats
    - (Optionally, print out what the failure was)

General Layout – Multiple Ways

- Start the **`while`** loop by
  - Getting the user's input
  - Don't assume the requirements have been met
    - (Do not change the Boolean flag at the start of the loop)
- Check each way of satisfying the requirements
  - If one of the ways satisfies the requirements, change the Boolean flag so the loop doesn't repeat

Time for…

LIVECODING!!!

Announcements

- HW 3 is out on Blackboard now
  - Complete the Academic Integrity Quiz to see it
  - Due by Friday (Feb 24th) at 8:59:59 PM
- Midterm is in class, March 15th and 16th
  - Week before Spring Break
  - Survey #1 will be released that week as well