

# **Strings, Lists and Tuples**

## **Intro to Functions**

**CS 8: Introduction to Computer Science, Winter 2019**  
**Lecture #3**

Ziad Matni, Ph.D.  
Dept. of Computer Science, UCSB

# A Word About Registration for CS8

---

- This class is **FULL**,  
& the waitlist is **CLOSED**.

# Administrative

---

- Lab01 – tomorrow
- Hw01 – due today
- Hw02 – due next week
- Modifications to class schedule
- Linux workshop
- Python IDLE

# Lecture Outline

---

- Strings & Operations on Strings
- Intro to Lists & Tuple
- Intro to Functions

**Yellow Band = Class Demonstration! ☺**

# Strings

---

- Collection of *characters*
- A string literal is enclosed in quotes
  - Use either double-quotes ("") or single quotes ('')

Examples:

```
name = "#JimboJones@UCSB? Wow!"  
nombre = 'Lisa Simpson!!'
```

# Special Characters in Strings

- What would you do if you wanted a string to be:  
I said "hello!"
- Answer: use the special character indicator \
  - The back-slash

Example:

```
message = "I said \"hello!\""
```

**Demo!**

# Strings as Objects

---

- Strings are **objects** of a Python *class* named **str**
- Lots of built-in functions work for string *objects*
- Class = an general “blueprint”
- Object = a particular “instant” of a class

# Operations on Strings

- **Concatenation**
  - Merging multiple strings into 1
  - Use the `+` operator
    - `"say my" + " " + "name"` will become `"say my name"`
- **Repetition**
  - Easy way to multiply the contents of a string
  - Use the `*` operator
    - `"ja " * 3` is `"ja ja ja "` (*why is there a space at the end?*)

**Demo!**

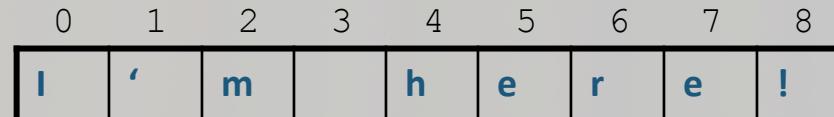
# Indexing

- Every character in a string has an index associated with it

0	1	2	3	4	5	6	7	8
I	'	m		h	e	r	e	!

- In Python, indexing always starts at **0**.
  - So the 1<sup>st</sup> character in the string is character #0
  - Indexing is called out with square brackets  $[n]$

# Indexing



- If `name = "I'm here!"` then:

`name[0] = "I"`

`name[3] = " "`

`name[5] = "e"`

`name[15]` is undefined (error)

# Indices and Slices

---

- To slice a string into a smaller string, use **[*i:j*]**
  - Where *i* = starting index, *j* = ending index (NOT included)
  - Example: "Gaucho"[2:4] is "uc"
- Combinations are possible!
  - Example, what does this spell out?  
**( ("o" + "Gaucho"[2:5] + " ") \* 3 ) + "!"**

# Exercise

- What is the value of s after the following code runs?

```
s = 'abc'  
s = 'd' * 3 + s  
s = s + e* 2
```

- A. ‘abcd3e2’
- B. ‘abcdddabc’
- C. ‘dddabcee’
- D. ‘abcdddabce2’
- E. Error

# Lists

---

- A **list** is a collection of multiple values
  - Similar to how a str is a collection of characters
- Note: In Python, lists can be of *heterogenous*
  - Of different types (i.e. ints or strings or etc...)
- Lists can also have duplicate values
- Lists are ***mutable***
  - The elements of a list can be **modified**

# Example of Lists

```
NameList = ["Abby", "Bruce", "Chris"]  
Student = ["Jill Jillson", 19, 3.7, "F"]
```

NameList and Student are variables of type **list**

- You can call up list elements by indexing the list  
Example: NameList[0] = "Abby"

More on lists later...

# Tuples

- Tuples are a variable type that's very similar to lists, except they are *immutable*!
  - That is, once they're set, they cannot change
- Example:  
collection = (1, 2, "buckle my shoe")

More (but not much more) on tuples later...

# Functions

---

# Procedural Abstraction: The Function

- A “black box” – a piece of code that can take inputs and gives me some expected output
- A **function**, for example, is a kind of procedural abstraction

25 → **Square Root Function** → 5

- What's happening inside the function?
- Doesn't matter, as long as it works!!

# Functions

---

- A function does “something” to one/several input(s) and sends back one/several output(s)
  - Always has *parentheses* to “carry” the inputs
- Example: the `sqrt()` function (square root)
  - With an input of 25, I expect an output of 5
  - That is, **`sqrt(25)`** will give me 5

# More About Functions

---

- Definition:  
*“Self contained” modules of code that accomplish a specific task.*
- Functions have **inputs** that get **processed** and the function often (although not always) **“returns”** an **output** (result).
- Functions can be **“called from”** the main block of a program
  - Or from inside other functions!

# More About Functions

---

- A function can be used over and over again.

- Example:

Consider a function called “*distance*” that returns the value of the distance between a point w/ coordinates (a, b) and the Cartesian origin (0, 0)

$$\textit{distance}(a, b) = \textit{square root of}(a^2 + b^2)$$

# Defining Your Own Function

- To define a function in Python, the syntax is:

```
def functionName (list of parameters):  
    # a block of statements appear here  
    # all of them must be indented (with tabs)
```

- **def** – a mandatory keyword that defines a function
- **functionName** – any legal Python identifier (e.g. myLittleFunction)
- **( ):** – mandatory set of parentheses and colon
- ***list of parameters*** – object names
  - **Local** references to objects (i.e. raw data or variables) that are passed into the function
- e.g. **def myLittleFunction(pony1, pony2, 3.1415):**

# Example Definition

---

```
# My first function! Yay!
def dbl(x):
    """This function returns double its input x"""
    print("Doubling the number to:", x)
    return 2*x    # I need to "return" the result
```

**Let's try it out!**

# FUNCTION RULES!

```
# My first function! Yay!
```

```
def dbl(x):
```

*Function header*

*x is the input parameter (also called argument)*

```
    """This function returns double its input x"""
```

*Function body*

```
        print("Doubling the number to:", x)
```

```
        return 2*x      # I need to "return" the result
```

Indentation: **VERY IMPORTANT**

Achieved with a tab character or just spaces

All the lines in the function body are indented from  
the function header; and all to the **same** degree

*docstring: a comment that becomes part of Python's built-in help system!*

*With each function be sure to include one that:*

- a) *describes overall what the function does, and*
- b) *explains what the inputs mean/are*

# More Example Definitions

```
# This function calculates the distance between (a,b) and (0,0)
def distance(a, b):
    x = a**2      # Note the tab indent!!!
    y = b**2      # Recall ** means "to the power of"
    z = (x + y) ** 0.5
    return z     # I need to "return" the result
```

**!!! Alternatively !!!**

```
def distance(a, b):
    return ( (a**2) + (b**2) ) ** 0.5
```

**Let's try it out!**

# Flow of Execution of a Function

- When you call a function, you have to use its name and its parameter(s) *just like they were defined*
  - Example: to call the dbl function on 21, you'd have to call it like this:  
**dbl(21)**
- When you call a function, Python executes the function starting at the first line in its body, and carries out each line in order
  - Though some instructions cause the order to change... more soon!

# Parameters are Specialized Variables

- When you call a function, the value you put in parenthesis gets put into a special part of computer memory that's labeled with the name of the parameter and is available for use within the function
- Example: in **dbl(x)**, the var. **x** can be used several times within that function

# What if There are Multiple Parameters??

- When you call a function, the values you put in parenthesis have to be in the order in which they are listed in the definition!

- Example:

```
def subtract(m, n):  
    return m - n
```

When you call this function to do a subtraction of  $5 - 99$ , then:

**m has to be 5 and n has to be 99**

So, it's called as:

subtract(5, 99)

*i.e. not* subtract(99, 5)

# What About... NO Parameters?!

- Sure, you can do that!

- Example:

```
def fortyTwo():  
    return 42
```

*All this function does is return the number 42 to whoever called it!*

*Which way should we call it?*

*fortyTwo  
fortyTwo()*

# Wow. Functions are Cool.

## *Can They CALL EACH OTHER????*

**Yes!!!!!!!!!!!!!! Careful that you get the order correct...!**

```
def halve( x ):  
    """ returns half its input, x """  
    return div(x, 2)
```

```
def div( y, x ):  
    """ returns y / x """  
    return y / x
```

**Let's try it out!**

**What happens when I say:**

```
>>> halve( 85 )
```

- A. I get 42
- B. I get 42.5
- C. 0
- D. 0.02352 (i.e., 2 divided by 85)

# YOUR TO-DOS

---

- Finish reading **Chapter 2**
- Start reading **Chapter 3**
- Start on **HW2** (due next **Monday**)
- Do **Lab1** (lab's tomorrow!)
  
- Embrace randomness

</LECTURE>