

CS 5001

LECTURE 5

SEQUENCED COLLECTIONS STRINGS, TUPLES & MORE LISTS THE WHY OF DESIGN

KEITH BAGLEY

FALL 2023

Northeastern University
**Khoury College of
Computer Sciences**

440 Huntington Avenue ■ 202 West Village H ■ Boston, MA 02115 ■ T 617.373.2462 ■ khoury.northeastern.edu

AGENDA

- Course Check-in
 - Lists & Loops Review
 - Slices Review
 - The Why of Design
 - Meta-data: Let's code the Gradekeeper v2 from last week
 - Debugging your code
 - Strings
 - Lists & Strings are Sequences
 - Tuples
 - Enumerate (and for, one more time)
 - HW 5 Insights
 - Q & A
-

CHECK-IN: COURSE MAJOR MILESTONE

Congratulations! After this lecture, you've learned all the tools you need to tackle some really interesting & complex problems

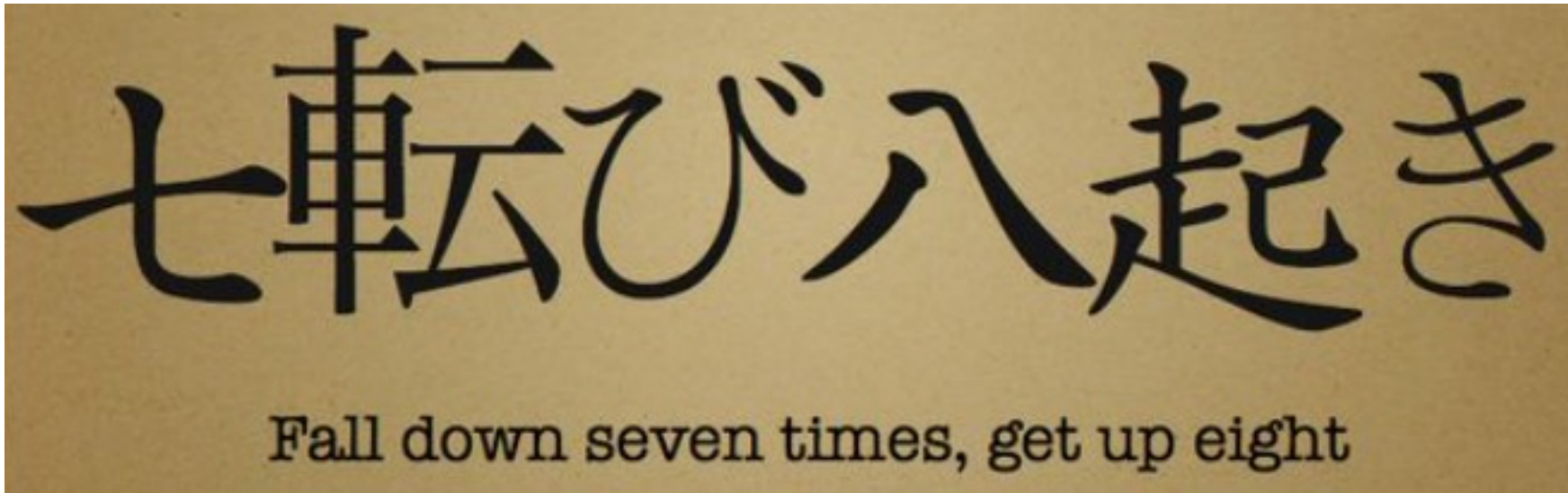


Image from: <https://www.oprah.com/world/women-climbing-mount-everest-hiking-mt-everest>

QUOTE OF THE WEEK

“[If you] Fall down seven times get up eight”

— Japanese Proverb

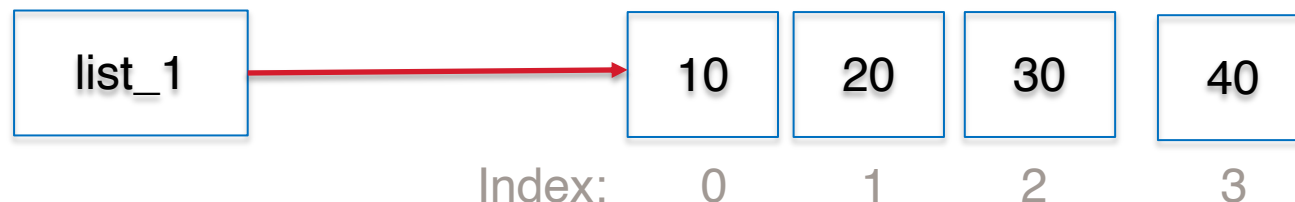


REVIEW: LISTS

- List elements can be accessed by index, using the name
- Created using the [] operator OR the list() function



```
>>> list_1 = [10, 20, 30, 40]    # a list of 4 integers
>>> keith_list = list()         # create an empty list using list function
>>> your_list = []              # create empty list using shorthand
```



REVIEW: HETEROGENEOUS LISTS

- The elements of a list don't have to be the same type.

Example:

```
>>> mixed_list = ['spam', 2.0, 5, [10, 20]]
```

`mixed_list` contains a string, a float, an integer, and another list.

- A list within another list is called a NESTED List.
- A list that contains no elements is called an empty list:

```
>>> empty_list = []
```

REVIEW: BY INDEX/POSITION OR VALUE

```
food = ["grapes", "apples", "snickers"]
i = 0
while i < len(food):
    food[i] = "good" + food[i]
    i = i + 1
print(food)
```

```
food = ["grapes", "apples", "snickers"]
for i in range(len(food)):
    food[i] = "good" + food[i]
print(food)
```

```
food = ["grapes", "apples", "snickers"]
for each in food:
    each = "good" + each
print(food)
```

```
lectures/lecture_code/food.py
['goodgrapes', 'goodapples', 'goodsnickers']
['goodgrapes', 'goodapples', 'goodsnickers']
['grapes', 'apples', 'snickers']
>>>
```



By position gives a reference
By value copies the value
Keep this in mind when we
talk about functions later...

LIST SLICES

- A List Slice is a subset of an existing list
 - Use the colon operator to specify what portion of the list you want



```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']  
>>> t[1:3] # element with index 3 is NOT  
included  
['b', 'c']
```

```
>>> t[:4] # start with the beginning till  
element of index 3  
['a', 'b', 'c', 'd']
```

```
>>> t[3:] # start with the element of  
index 3, till the end  
['d', 'e', 'f']
```

```
>>> t[: ] # the slice is the copy of the  
whole list  
['a', 'b', 'c', 'd', 'e', 'f']
```


EXERCISE FROM LAST WEEK: LET'S DO IT TOGETHER

Write a function called chop that takes a list, modifies (mutates) it by removing the first and last elements. The function returns nothing. For example:

```
>>> t = [1, 2, 3, 4]
```

```
>>> chop(t)
```

```
>>> t
```

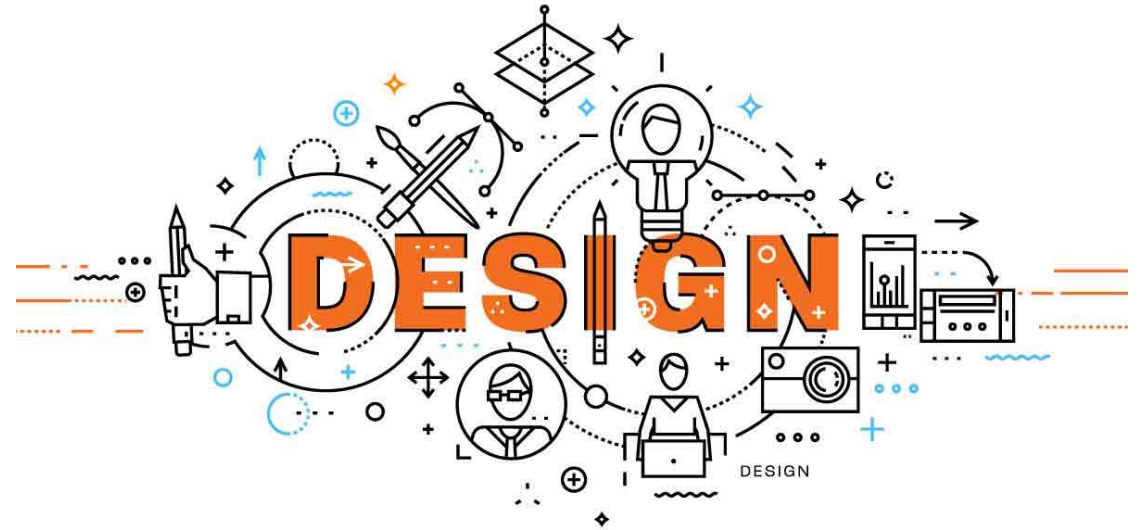
```
[2, 3]
```

THE “WHY” OF DESIGN

- We're at the point in the class where we will shift to asking you to build larger, more complex solutions

THE “WHY” OF DESIGN

- Some design fundamentals
 - Procedural Decomposition
 - Computational thinking & algorithms
 - What is the data (and type of data) that I need to know/remember?
 - What are the functions?
What is the IPO I need to manage?



THE “WHY” OF DESIGN – LET’S DO IT!

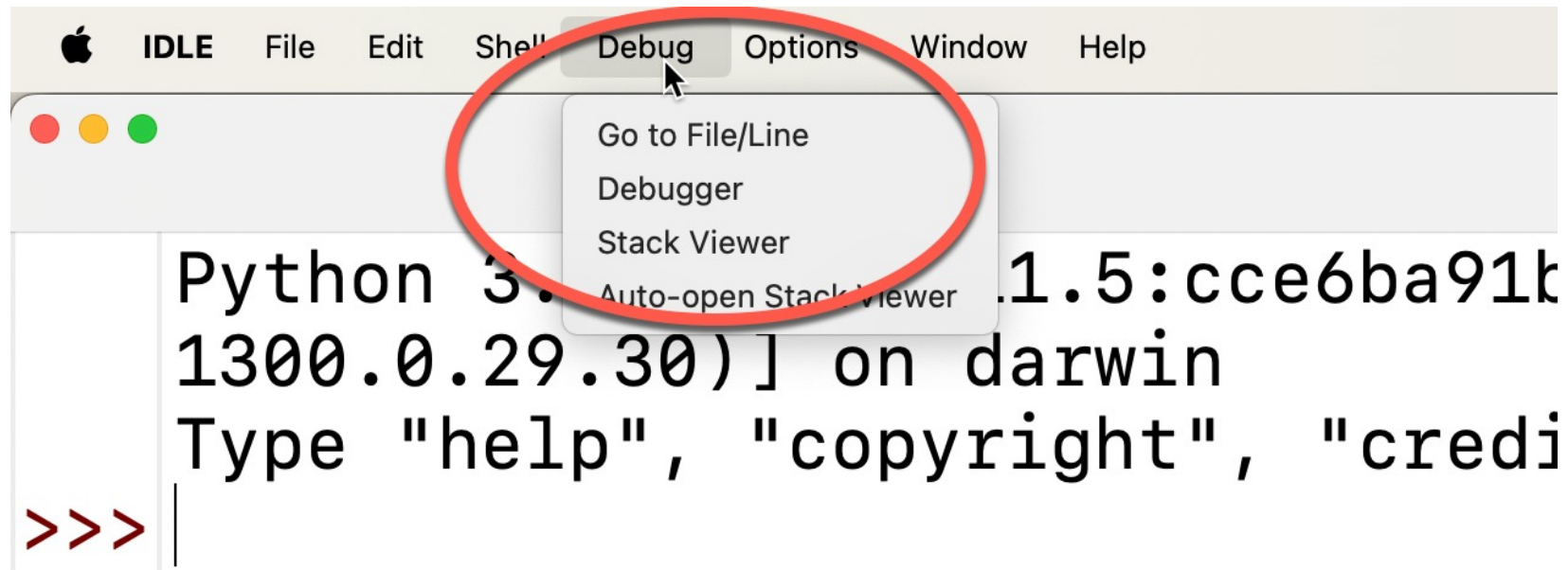
- Let’s examine the Grade program, using metadata
 - Think first, then do
 - What is our data structure and schema?
 - Can we use existing code or concepts?

DEBUGGING

- How many of you always write perfect, error-free code?
 - All the time?
 - No, really – no mistakes in syntax, no logic errors, no missing branch-flows?
 - Even A.I. makes mistakes – let's talk about how to work through that via debugging

DEBUGGING

- Easy, but not scalable: print() statements to inspect various variables and application “state”
- Less easy, but not difficult and much more powerful – use a debugger



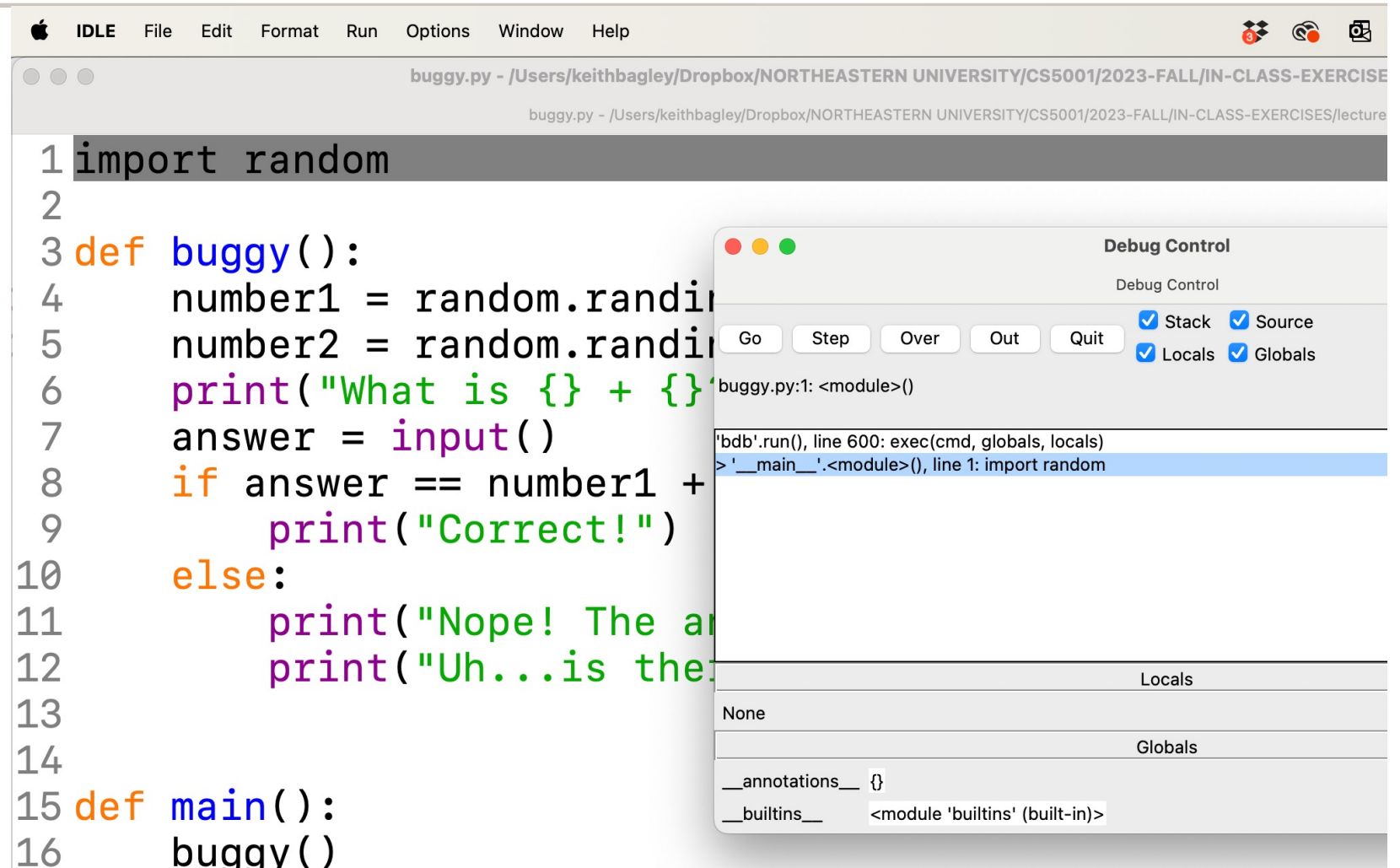
DEBUGGING

- You can use simple print statements for a quick view into the values at various points in your code.
- A debugger is more extensive, and allows you to inspect variables, step one line at a time through your code, and watch as variable values change
- For more complex applications, use a debugger! Get experience with using one now – you'll definitely need it by the end of the course, and for the rest of the MS program!



DEBUGGING

- Let's practice with the buggy.py file!



The screenshot shows a Python IDE window titled 'buggy.py - /Users/keithbagley/Dropbox/NORTHEASTERN UNIVERSITY/CS5001/2023-FALL/IN-CLASS-EXERCISE'. The code in the editor is as follows:

```
1 import random
2
3 def buggy():
4     number1 = random.randint(1, 10)
5     number2 = random.randint(1, 10)
6     print("What is {} + {}".format(number1, number2))
7     answer = input()
8     if answer == number1 + number2:
9         print("Correct!")
10    else:
11        print("Nope! The answer is {}".format(number1 + number2))
12        print("Uh...is that all?")
13
14
15 def main():
16     buggy()
```

A 'Debug Control' window is open, showing the following options:

- Go
- Step
- Over
- Out
- Quit
- ☒ Stack
- ☒ Source
- ☒ Locals
- ☒ Globals

The 'Locals' pane shows:

Locals
None

The 'Globals' pane shows:

Globals	
__annotations__	{}
__builtins__	<module 'builtins' (built-in)>

SEQUENCES

- Strings, Lists, Tuples

STRINGS

- Strings are sequences of characters
- Similar to Lists, but Strings are IMMUTABLE

The bracket operator: access the characters one at a time

```
>>> fruit = 'banana'
```

```
>>> letter = fruit[1]
```

```
>>> letter
```

```
'a'
```

0	1	2	3	4	5
b	a	n	a	n	a



STRINGS

- **len()** is a built-in function that returns the number of characters in a string:

```
>>> fruit = 'banana'
```

```
>>> len(fruit)
```

```
6
```

```
>>> lego = 'storm trooper'
```

```
>>> len(lego)
```

```
13
```



STRINGS



- Hold on...wait a minute...
- Are we in a time loop? Or Python infinite loop?
- Didn't we talk about this stuff last week?

LISTS

- A List is a named collection of data
 - Similar to the to-do and shopping lists you might use
 - Allows us to refer to a group of data values by ONE name

```
>>> list_1 = [10, 20, 30, 40] # a list of 4 integers
>>> list_2 = ['yummy', 'rummy', 'tummy'] # a list of 3 strings
```

THINGS to DO

named collection of data to the to-do and shopping lists you might use

```
[10, 20, 30, 40] # a list of 4 integers
['yummy', 'rummy', 'tummy'] # a list of 3 strings
```

THINGS to DO

data shopping lists up of data

```
] # a list of 4 integers
['yummy', 'rummy', 'tummy'] # a list of 3 strings
```

THINGS to DO

is a named collection of data similar to the to-do and shopping lists you might use

```
_1 = [10, 20, 30, 40] # a list of 4 integers
_2 = ['yummy', 'rummy', 'tummy'] # a list of 3 strings
```

LISTS

- List elements can be accessed by index, using the name
- Created using the [] operator OR the list() function

```
>>> list_1 = [10, 20, 30, 40] # a list of 4 integers
```

THINGS to DO

elements can be accessed by index, using the name

```
[10, 20, 30, 40] # a list of 4 integers
```

THINGS to DO

by index, OR the list()

```
list_1 = [10, 20, 30, 40] # a list of 4 integers
```

THINGS to DO

elements can be accessed by index, using the name

```
list_1 = [10, 20, 30, 40] # a list of 4 integers
```

ANATOMY OF A LIST

```
>>> list_1 = [10, 20, 30, 40] # a list of 4 integers
```

ANATOMY OF A LIST

```
list_1 = [10, 20, 30, 40] # a list of 4 integers
```

ANATOMY OF A LIST

```
list_1 = [10, 20, 30, 40] # a list of 4 integers
```

ANATOMY OF A LIST

```
list_1 = [10, 20, 30, 40] # a list of 4 integers
```

STRINGS



- Yes!
- Strings, Lists (and Tuples) are all Sequences
- They share a common protocol
 - i.e. there are common functions and operators that can be applied to them all – the Sequenceable protocol
 - This allows for uniformity in how we work with every sequence

STRINGS

- Familiar operations apply to Strings

How do you get the last letter of a string?

```
>>> m = len(fruit)
```

```
>>> last = fruit[      ]
```

Negative indices:

- The expression **fruit[-1]** yields the last letter
- The expression **fruit[-2]** yields the second to last

STRINGS: TRAVERSAL

- Familiar operations apply to Strings

for and while loops work with strings

```
fruit = "banana"
```

```
index = 0
```

```
while index < len(fruit):
```

```
    letter = fruit[index]
```

```
    print(index, letter)
```

```
    index = index + 1
```

```
fruit = "banana"
```

```
for letter in fruit:
```

```
    print(letter)
```

STRINGS: CONCATENATION

In Robert McCloskey's book *Make Way for Ducklings*, the names of the ducklings are **Jack, Kack, Lack, Mack, Nack, Ouack, Pack, and Quack.**

My colleague Sarah (San Jose professor) attempted to print these names in order using the following loop:

```
prefixes = 'JKLMNOPQ'  
suffix = 'ack'
```

What did Sarah need to do to print the names correctly using a loop?

STRINGS: CONCATENATION

In Robert McCloskey's book *Make Way for Ducklings*, the names of the ducklings are **Jack, Kack, Lack, Mack, Nack, Ouack, Pack, and Quack**.

Sarah can do this:

```
prefixes = 'JKLMNOPQ'
suffix = 'ack'
for letter in prefixes:
    if letter == 'O' or letter == 'Q':
        print(letter + 'u' + suffix)
    else:
        print(letter + suffix)
```

```
prefixes = 'JKLMNOPQ'
suffix = 'ack'
for letter in prefixes:
    if letter == 'O' or letter == 'Q':
        print(letter + 'u' + suffix)
    else:
        print(letter + suffix)
```

Output

```
_code/lecture_:
Jack
Kack
Lack
Mack
Nack
Ouack
Pack
Quack|
```

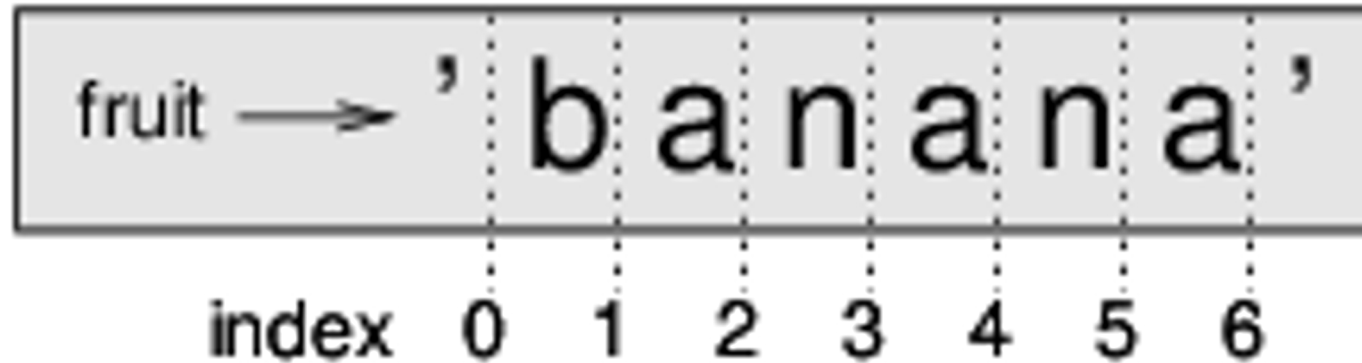
STRINGS: WE CAN SLICE THEM TOO

- Familiar operations apply to Strings

```
>>> fruit[2:4]  
'na'
```

```
>>> fruit[:3]  
'ban'
```

```
>>> fruit[3:]  
'ana'
```



STRINGS CANNOT BE CHANGED

strings are **immutable** (cannot change an existing string)

```
>>> greeting = 'Hello, World!'
```

```
>>> greeting[0] = 'J'
```

TypeError: 'str' object does not support item assignment

[Question] How can you make it a “Jello, World!”?

MAKE IT JELLO WORLD

```
>>> greeting = 'Hello, World!'
```

1. Get a string slice: 'ello, World!'

```
>>> str_slice = greeting[1: ]
```

2. Create a new string by concatenating "J" and str_slice:

```
>>> new_greeting = "J" + str_sclice
```

COMMON STRING METHODS

upper ()	Converts a string into upper case
title ()	Converts the first character of each word to upper case
find ()	Searches the string for a specified value and returns the position of where it was found
count()	Returns the number of times a specified value occurs in a string
replace()	Returns a string where a specified value is replaced with a specified value

All string methods returns new values. They do not change the original string.

LISTS & STRINGS – REMEMBER!

- **A string** is a sequence of characters and **a list** is a sequence of values,
- A list of characters is not the same as a string.
- To convert from a string to a list of characters, use `list()`

```
>>> s = 'spam'
```

```
>>> t = list(s)      # The list function breaks a string into individual letters.
```

```
>>> t
```

```
['s', 'p', 'a', 'm']
```

LISTS & STRINGS - SPLIT()

If you want to break a string into words, you can use the `split` method:

```
>>> s = 'Five Guys Burgers and Fries'
```

```
>>> t = s.split()
```

```
>>> t
```

```
['Five', 'Guys', 'Burgers', 'and', 'Fries']
```

LISTS & STRINGS - SPLIT()

An optional argument called a **delimiter** specifies which characters to use as word boundaries. The following example uses a hyphen as a delimiter:

```
>>> s = 'spam-spam-spam'
```

```
>>> delimiter = '-'
```

```
>>> t = s.split(delimiter)
```

```
>>> t
```

```
['spam', 'spam', 'spam']
```


LISTS & STRINGS - JOIN()

- `join` is the inverse of `split`.
- `join` is a string method (so you have to invoke it **on the delimiter** and pass the list as a parameter)

```
>>> t = ['pinning', 'for', 'the', 'fjords']  
>>> delimiter = '  
>>> s = delimiter.join(t)  
>>> s  
'pinning for the fjords'
```

OPERATORS

- 1) The bracket operator indexes an element:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
```

```
>>> print t[0]
```

```
'a'
```

- 1) The slice operator selects a range of elements.

```
>>> print t[1:3]
```

```
('b', 'c')
```

- 1) What if you try this:

```
>>> t[0] = 'A'
```



EXERCISE: LET'S DO THIS TOGETHER

Most computers use *Binary* (base-2) as their underlying “language” to represent our programs and data. Humans tend to think in *decimal* (base-10).

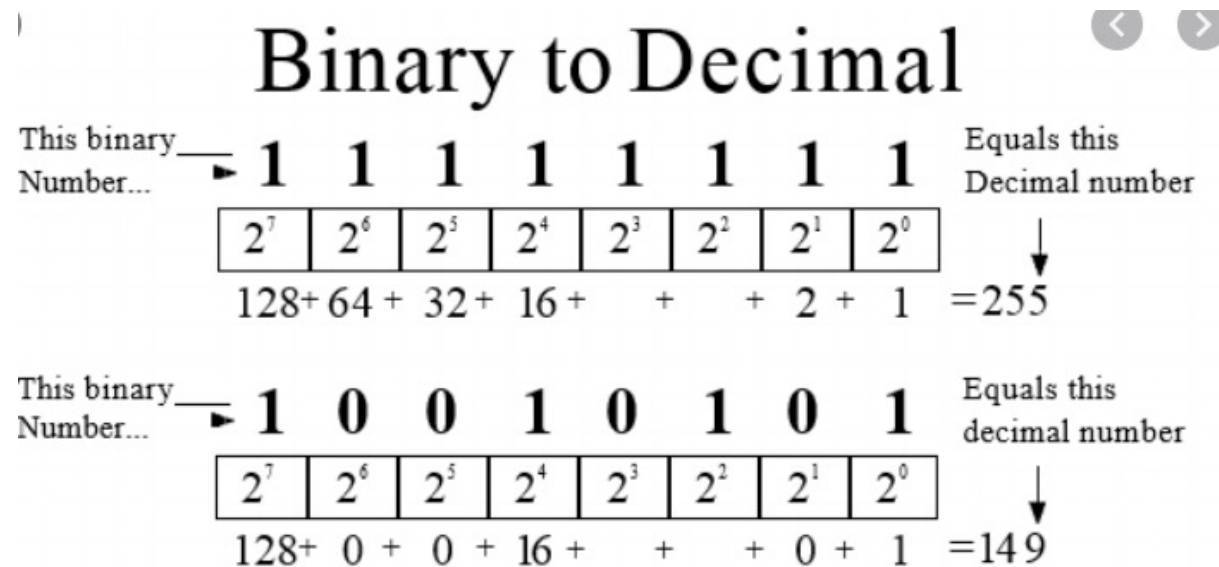
Let's write a **function** that takes a binary string of 1's and 0's as **input** and **returns** the decimal equivalent

'001' -> 1

'100' -> 4

'101' -> 5

Etc.



YOU TRY: PALINDROME CHECKER

- A palindrome is a word, phrase or sequence that is spelled the same way forwards and backwards. For example, racecar is a palindrome (so is 'LOL' in text speak)
- Write a function called `is_palindrome(word)` that takes a word (a string) and returns True if the word/phrase is a palindrome, or False if it is not.
 - Your function should use the string `.replace()` to substitute any blank spaces with the empty string, thereby removing spaces. For example: 'race car' becomes 'racecar'
- You are free to use string slices, loops, lists, or whatever. In fact, there are multiple approaches to this problem so if you have time, try to solve this using multiple approaches

TUPLES



- Wait! So these Tuple thing-a-ma-jigs use the same operators as Lists & Strings?
- Yes!
- Strings, Lists (and Tuples) are all Sequences
- One more time: They share a common protocol
 - i.e. there are common functions and operators that can be applied to them all – the Sequenceable protocol
 - This allows for uniformity in how we work with every sequence
- *Schweet!*

TUPLES ARE IMMUTABLE

```
>>> t[0] = 'A'
```

```
TypeError: object doesn't support item assignment
```

- You can't modify the elements of a tuple (it is immutable),
- but you can replace one tuple with another:

```
>>> t = ('A',) + t[1:]
```

```
>>> print t
```

```
('A', 'b', 'c', 'd', 'e')
```

TUPLES ARE LIKE LISTS BUT IMMUTABLE

A tuple is a sequence of values of any type:

- indexed by integers
- tuples are immutable (while lists are mutable)

Syntactically, a tuple is a comma-separated list of values:

```
>>> t = 'a', 'b', 'c', 'd', 'e'    # this is fine
```

it is common to enclose tuples in parentheses:

```
>>> t = ('a', 'b', 'c', 'd', 'e')  # this is more common
```

CREATING A TUPLE

To create a tuple with a single element, you have to include a final comma:

```
>>> t1 = 'a',          # include a final comma to make a tuple
>>> type(t1)           # it is the commas that make the tuple not the ()!
<type 'tuple'>
```

A value in parentheses is not a tuple:

```
>>> t2 = ('a')
>>> type(t2)
<type 'str'>
```

TUPLE ASSIGNMENT

Tuples can be used to “shorthand” a multiple-value assignment

- Values are “unpacked” in proper order: left-to-right
- Swap is a good example, but also useful for returning multiple values from functions

```
>>> one = 1
>>> two = 2
>>> one, two = two, one
>>> print(f"one = {one} two = {two}")
one = 2 two = 1
>>> |
```

```
def menu(message, options='123'):
    '''
    Function:    menu
    Description: This function accepts a query msg and a set of
                  acceptable options as input. Returns answer from the
                  user AND a flag indicating if the answer was within
                  the acceptable options

    Params:      message - string message for user interaction
                  options (optional) - string that gives us the set of
                  allowable possibilities.

    Returns:     a Tuple. The answer and True/False depending on if user
                  choice is within the options specified
    '''

    answer = input(message)
    if len(answer) == 0:
        return answer, False # user pressed return w/o an answer
    answer = answer[0].upper() # convert to uppercase for comparison
    if answer in options.upper(): # compare first letter if they enter more
        return answer, True
    return answer, False # return a tuple with answer & success flag
```

“OLD SCHOOL” VS. TUPLE ASSIGNMENT

A temporary variable is used in conventional assignments:

```
>>> temp = a
```

```
>>> a = b
```

```
>>> b = temp
```

Tuple Assignment makes this easy:

```
>>> a, b = b, a
```

SWAPPING THE VALUES OF TWO VARIABLES

```
>>> a, b = b, a
```

- The left side is a tuple of variables
 - The right side is a tuple of expressions
 - Each value is assigned to its respective variable.
 - All the expressions on the right side are evaluated before any of the assignments.
-

TUPLE ASSIGNMENTS

```
>>> addr = 'monty@python.org'
>>> uname, domain = addr.split('@')
```

Using Tuple Assignments: The right side can be any kind of sequence (string, list or tuple)
In this example:

- The return value from split is a list with two elements
- The first element is assigned to uname, the second to domain.

```
>>> print uname
monty
>>> print domain
python.org
```

TUPLE ASSIGNMENT – PITFALLS!

The number of **variables** on the left and the number of **values** on the right have to be the same:

```
>>> a, b = 1, 2, 3
```

```
ValueError: too many values to unpack
```

ENUMERATE

- Now that we've covered Tuples...
 - Let's briefly talk enumerate()
 - Useful constructs for iteration
 - Supported by many languages
 - Helps you when you want the best of for-each (by value) AND [index] (by position) access

ENUMERATE

```
>>> p = [1, 2, 3, 4]
>>> p
[1, 2, 3, 4]
>>> for index, value in enumerate(p):
    print(index, value)
```

Get the index and value as a Tuple;
Unpack into individual variables

```
0 1
1 2
2 3
3 4
>>> for each in enumerate(p):
    print(each)
```

Or..
Use the tuples as they are

```
(0, 1)
(1, 2)
(2, 3)
(3, 4)
```

DESIGN THOUGHTS: APPLYING WHAT YOU KNOW

- At this point, you've gotten all you need to start rolling up your sleeves and solve some pretty interesting computer science and software engineering problems
- The rest of the course will continue to introduce new material BUT we'll also focus on applying what you've already learned for good design & implementation
- The upcoming HW5 is an example – fewer individual pieces and more “put what you know to use to solve this larger problem”

EXAMPLE: DESIGN THOUGHTS - (SYNC'D LISTS)

- Synchronized Lists (better known as Associative Arrays outside of Python) work okay and can be one option
- But can be a maintenance/management problem

EXAMPLE: DESIGN THOUGHTS - (SYNC'D LISTS)

- What do we mean by a maintenance/management problem?
- Consider a program to track movie reviews
 - How did Greenbook do?
 - Find in 1st list
 - Get the same position in 2nd list
 - Add a new movie
 - Don't forget to add to the second list too, or else!
 - What if we want to track more characteristics
 - Add more lists to track those qualities OR add a list with multi-faceted information that we need to remember index/positions for

SYNC'D LISTS

- Two lists can correspond (Associative Arrays)

movies[i]

'Aquaman '	'Black Panther '	'Greenbook '	'Mary Poppins '
0	1	2	3

ratings[i]

5	5	4	1
0	1	2	3

SYNC'D LISTS

- Two lists can correspond (Associative Arrays)

movies[i]

'Aquaman '	'Black Panther '	'Greenbook '	'Mary Poppins '
0	1	2	3

ratings[i]

5	5	4	1
0	1	2	3

OTHER APPROACHES

Next week we'll introduce the dictionary data structure as an “alternate form” of managing aggregated data

Key idea:

There are often multiple solution paths. Sometimes one is better (or more efficient) than others, but other times it is a case of “six in one hand, half-dozen in the other”. Design is about “trade-off” decisions. Keep this in mind as you embark on HW5

LAUNCH INTO LAB

An **acrostic** is a form of writing where a set letter of each line spells out a word or message. Often acrostics (especially poems) use the first letter to spell out their messages, but other “columns” in the text may be used.

For this lab, we'll create an acrostic reader. Given the poem and dream data in [acrostics_data.py](#), write a function called `print_acrostic()` that takes a list that contains two elements:

1. a string containing the poem/dream, and
2. an integer that specifies which “column” the acrostic is to be read from.

and prints the acrostic to the screen. For this lab, you're only given two text sources, but you should write your code in such a way that it can handle an arbitrary number of text/column pairs. An example `main()` might look like this:

```
def main():  
    acrostics = [[POEM, 0], [DREAM, 3]]  
    for each in acrostics:  
        print_acrostic(each)
```

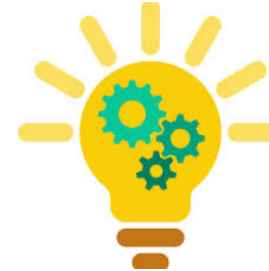
The poem is a message from *Northeastern* and uses the first character of each line (index 0); the dream is a *message from New York* and uses the 4th character of each line. Import (or copy) the lists from `acrostics_data.py` file [HERE](#)

For example, given POEM is this

```
POEM = "Intensive technologies croon.\n\nLocal bandits lesson.\n" \  
        "Old notebooks emote.\n\nVideos intersect.\n" \  
        ... # etc.
```

The acrostic at position zero would spell out a phrase: "I LOV..."

"Blank" lines created by two newline characters in a row (e.g. '\n\n') represent a space between words in the phrase (which is why the above example is "I LOV" rather than "ILOV")



What do you need to know & remember?



What do you need to do?

FACES OF C.S.: LIXIA ZHANG



Is a Professor of Computer Science at UCLA Her expertise is in computer networks she helped found the Internet Engineering Task Force and **designed the Resource Reservation Protocol**

Zhang grew up in northern China, where she worked as a tractor driver on a farm when the Cultural Revolution closed the schools. She earned a master's degree in electrical engineering at California State University and completed her doctorate at MIT in 1989

Source: https://en.wikipedia.org/wiki/Lixia_Zhang

PROVOKING THOUGHTS FOR THE WEEK

APPLE POLICY TECH

Here's how the FBI managed to get into the San Bernardino shooter's iPhone

An Australian firm helped hack into the device, starting with a Lightning port exploit

By Mitchell Clark | Apr 14, 2021, 3:58pm EDT

f t SHARE



Are there suitable instances where companies should help the government hack private citizens' data?

<https://www.theverge.com/2021/4/14/22383957/fbi-san-bernadino-iphone-hack-shooting-investigation>

Q & A

- Questions?
-

THANKS!

- Stay safe, be encouraged, & see you next week!

