

The background is a solid pink color. It is decorated with various hand-drawn geometric shapes in white and black. These include a dashed line in the top left, a white triangle in the top center, a black zigzag line in the top right, a white circle in the top right, two parallel black lines in the top right, a white triangle in the top right, a large black circle in the bottom right, a white circle in the bottom right, a black plus sign in the bottom left, a white circle in the bottom left, a white triangle in the bottom left, and a black plus sign in the bottom left.

Welcome!

We'll get started shortly ...



CS 49 Section


Week 10

Surajit A Bose





Agenda

- Logistics and check-ins
 - Review of lecture concepts
 - Single values vs containers
 - Immutable vs mutable objects
 - Creating lists and accessing list elements
 - List methods
 - Section Problems
 - [List practice](#)
 - [Index game](#)
 - [Heads up!](#)
- 




Logistics





How to get hold of me / get help+

- The [class forum](#). Feel free not only to ask, but also to answer questions!
 - Surajit's office hours:
 - Fridays 12 noon–1p, directly after section
 - By appointment on [Zoom](#)
 - [Lane's office hours](#)
 - Canvas inbox or Pronto inbox for Lane
 - Canvas inbox (preferred) or Pronto for Surajit
 - [Sina's support section](#), Fridays 1p–2p on [Zoom](#)
 - Email bozesurajit@fhda.edu, 24 hr turnaround
 - [Online](#) or [in-person](#) tutoring via the STEM center (Room 4213)
 - The section [GitHub repo](#) has lecture and section slides and solutions
- 




Lecture Review: Containers






Atomic types vs Containers

- In Python, **int**, **float**, **Boolean**, or **None** are all **atomic** types
 - A value of those types is a single value
 - Python also has **container** types
 - Containers are what they sound like
 - They can be empty, or they can hold any number of values
 - They are **collections** of elements
 - We can access individual elements in the container
 - We can **iterate** (loop) over all the elements in a container
 - Surprise! We've been working with one container type already
 - Any guesses as to what?
- 



Atomic types vs Containers



- In Python, **int**, **float**, **Boolean**, or **None** are all **atomic** types
 - A value of those types is a single value
 - Python also has **container** types
 - Containers are what they sound like
 - They can be empty, or they can hold any number of values
 - They are **collections** of elements
 - We can access individual elements in the container
 - We can **iterate** (loop) over all the elements in a container
 - Surprise! We've been working with one container type already
 - **str**
- 

Atomic types vs Containers

- A string can be empty:

```
null_string = ''
```

- A string can be arbitrarily large:

```
big = 'This is supercalifragilisticexpialidocious'
```

- We can iterate over the string:

```
for char in big:  
    print(char)
```

- This will print 'T', 'h', 'i', 's', ' ', 's', 'u', etc. each on its own line
- We can get a string's length: **len(big)** will evaluate to 42

The background is a solid orange color. It is decorated with various hand-drawn geometric shapes in white and black. These include: a dashed line in the top left; a white triangle outline in the top center; a black zigzag line in the top right; a white circle outline in the top right; two parallel black lines in the top right; a white triangle in the top left; a black plus sign in the bottom left; a white circle outline in the bottom center; a white triangle outline in the bottom center; a black plus sign in the bottom center; a black circle outline in the bottom right; and a white circle in the bottom right.

Any Questions?



Lecture Review: Mutables



Immutable vs Mutable types

- All the types we've been looking at so far have been **immutable**
- By definition, immutable objects cannot be changed in memory; they can only be replaced with a different object
- All atomic types are immutable
- Strings are immutable:

```
hi = 'Hello'
```

```
# creates a string in memory
```

```
hi = 'Hello there'
```

```
# creates a new string
```

```
hi.append('!')
```

```
# this will not change the  
# string to 'Hello there!',  
# it will error out
```



Immutable vs Mutable types

- Python has several **mutable** container types
- These containers can be changed in memory without being replaced
- We can add elements to or remove elements from the container
- One useful mutable container type is **list**
- As with **str**:
 - A **list** can be empty or arbitrarily large
 - We can iterate over the elements of a **list**
- Unlike **str**:
 - We can mutate a **list**
 - Elements in a **list** do not all have to be the same type

The background is a solid orange color. It is decorated with various hand-drawn geometric shapes in white and black. These include: a dashed line in the top left; a white triangle in the top center; a black zigzag line in the top right; a white circle in the top right; two parallel black lines in the top right; a white triangle in the top right; a large black circle in the bottom right; a white circle in the bottom right; a black plus sign in the bottom left; a white circle in the bottom center; a white triangle in the bottom center; and a black plus sign in the bottom center.

Any Questions?



Lecture Review: Creating lists, accessing list elements



Creating a list

- A new, empty list can be created with a pair of square brackets:

```
my_list = []
```

- A new list can also be created with elements in it:

```
colors = ['red', 'blue']
```

- List elements are separated by commas
- Another way to create a new list is by adding two lists together:

```
more_colors = ['green', 'yellow']
```

```
colors_two = colors + more_colors
```

```
# colors_two is ['red', 'blue', 'green', 'yellow']
```

```
# Original lists colors, more_colors are unchanged
```


Accessing elements: List indices +

- An element's position in the list is its index
- Indices start at zero and go up to one less than the number of elements
- E.g., given **colors_two** = **['red', 'blue', 'green', 'yellow']**
 - **'red'** is at index 0
 - **'blue'** at index 1
 - **'green'** at index 2
 - **'yellow'** at index 3
- Access an element with the index in square brackets after the list name
shrek_color = colors_two[2] # *shrek_color* is 'green'

Accessing elements: List indices +

- We can also use negative indices
- The index of the last element is -1
- E.g., given `colors_two = ['red', 'blue', 'green', 'yellow']`
 - `'yellow'` is at index -1
 - `'green'` at index -2
 - `'blue'` at index -3
 - `'red'` at index -4
- Access an element with the index in square brackets after the list name
`gumby_color = colors_two[-3]` # *gumby_color* is `'blue'`

Checking membership and position

- `some_element in some_list` will return a **Boolean**
 - **True** if `some_element` is in `some_list`
 - **False** if `some_element` is not in `some_list`
- `some_list.index(some_element)` will return the index at which `some_element` is in `some_list`
- Given `colors_two = ['red', 'blue', 'green', 'yellow']`
 if `'blue' in colors_two`:
 `ind = colors_two.index('blue')` *# ind is 1*
- Can likewise check `some_element not in some_list`

The background is a solid orange color. It is decorated with various hand-drawn geometric shapes in white and black. These include: a dashed line in the top left; a white triangle in the top center; a black zigzag line in the top right; a white circle in the top right; two parallel black lines in the top right; a white triangle in the top right; a large black circle in the bottom right; a white circle in the bottom right; a white triangle in the bottom center; a dashed line in the bottom center; a black plus sign in the bottom left; a black circle in the bottom left; and a black plus sign in the bottom left.

Any Questions?



Lecture Review: Mutating lists



Adding to a list

+

- Add to the end of a list using `append()`

```
my_list = []
```

```
my_list.append('yo') # my_list is now ['yo']
```

```
my_list.append('hi') # my_list is now ['yo', 'hi']
```

- Add multiple elements to the end of a list with `extend(some_list)`

```
my_list.extend(['sup', 'howdy'])
```

```
# my_list is now ['yo', 'hi', 'sup', 'howdy']
```

- Add an element at a specific index with `insert(value, index)`

```
my_list.insert('hey', 2)
```

```
# my_list is now ['yo', 'hi', 'hey', 'sup', 'howdy']
```

Removing from a list

- Remove from the end of a list using `pop()`
my_list is now ['yo', 'hi', 'hey', 'sup', 'howdy']
`greeting = my_list.pop()`
greeting is 'howdy'
my_list is now ['yo', 'hi', 'hey', 'sup']
- Remove from a specific index by using `pop(index)`
`salutation = my_list.pop(1)`
salutation is 'hi'
my_list is now ['yo', 'hey', 'sup']

Replacing a specific element

+

- Replace an element at a specific index by assignment

```
# my_list is now ['yo', 'hey', 'sup']
```

```
my_list[1] = 'hiya'
```

```
# my_list is now ['yo', 'hiya', 'sup']
```

- Be clear about the distinction between
 - Removing an element at a specific index with **pop(index)**
 - Inserting a value at a specific index with **insert(value, index)**
 - Replacing an element at a specific index by assignment
- The third does not change the number of elements in the list

Any Questions?




Coding Challenge: Accessing elements, mutating lists





Challenge


- Say you have a list **fruits** with an arbitrary number of elements
 - You do not know whether **'raisin'** is in the list
 - If it is, you want to replace it with **'grape'**
 - If it is not, you want to add **'grape'** to the end of the list.
- 



Challenge

- Say you have a list **fruits** with an arbitrary number of elements
- You do not know whether **'raisin'** is in the list
- If it is, you want to replace it with **'grape'**
- If it is not, you want to add **'grape'** to the end of the list.

```
if 'raisin' in fruits:  
    ind = fruits.index('raisin')  
    fruits[ind] = 'grape'  
else:  
    fruits.append('grape')
```



The background is a solid orange color. It is decorated with various hand-drawn geometric shapes in white and black. These include: a dashed line in the top left; a white triangle in the top center; a black zigzag line in the top right; a white circle in the top right; two parallel black lines in the top right; a white triangle in the top right; a large black circle on the right edge; a white triangle in the bottom left; a black plus sign in the bottom left; a white circle in the bottom center; a white triangle in the bottom center; a dashed line in the bottom center; a black plus sign in the bottom center; a black circle in the bottom center; a white circle in the bottom right; and a black circle in the bottom right.

Any Questions?



Lecture Review: List methods





Other useful list methods

+



Say your intern was supposed to create a list **populations** whose elements represent the number of people in each of California's counties. You're not sure whether the intern has completed the work.

- Check if the list is empty: **if populations**
- Get the number of elements in the list: **len(populations)**

Let's say the length is 58, so you know your diligent intern is done.

- Get the total population of California: **sum(populations)**
- Get the population of the most populous county: **max(populations)**
- Get the population of the least populous county: **min(populations)**
- Get the mean population: **?**



Other useful list methods

+



Say your intern was supposed to create a list **populations** whose elements represent the number of people in each of California's counties. You're not sure whether the intern has completed the work.

- Check if the list is empty: **if populations**
- Get the number of elements in the list: **len(populations)**

Let's say the length is 58, so you know your diligent intern is done.

- Get the total population of California: **sum(populations)**
- Get the population of the most populous county: **max(populations)**
- Get the population of the least populous county: **min(populations)**
- Get the mean population: **sum(populations) / len(populations)**





Iterating over a list

- We can loop over a list element by element:

```
for name in names:  
    print(f'Hi, {name}')
```

- We can loop over a list index by index, if we need to change the list elements themselves:

```
for i in range(len(names)):  
    names[i] = names[i].upper()
```



Section problem: List Practice

<https://codeinplace.stanford.edu/cs49-w24/ide/a/fruitlist>





List Practice



- Create a list of fruits
- Print out the length of the list
- Add another fruit to the end of the list
- Loop over the list to print each element





Section problem: Index Game

<https://codeinplace.stanford.edu/cs49-w24/ide/a/indexgame>





Index Game

- Practice with list indices
- Please replace Line 17 with the following:

```
idx = random.randint(len(names) * -1, max_index)
```

- Let's play!
- 



Lecture Review: Passing lists as parameters



Lists as function parameters +

- We have seen that when a value of an immutable type is passed in as a function parameter, the function receives its own copy of the value:

```
def add_two(num):  
    num += 2  
    print(num)
```

```
def main():  
    num = 5  
    add_two(num)      # output: 7  
    print(num)        # output: 5
```

Lists as function parameters +

- But when a container of a mutable type, such as a list, is passed in as a function parameter, the function works with the original container:

```
def add_two_elements(my_list):  
    my_list.extend([6, 7])  
  
def main():  
    my_list = [1, 2, 3]  
    add_two_elements(my_list)  
    print(my_list)           # output: [1, 2, 3, 6, 7]
```


Lists as function parameters +

- This is true whether or not the called function uses the same name for the list as the calling function:

```
def add_two_elements(some_list):  
    some_list.extend([6, 7])
```

```
def main():  
    my_list = [1, 2, 3]  
    add_two_elements(my_list)  
    print(my_list)           # output: [1, 2, 3, 6, 7]
```

Any Questions?




Section problem: Heads Up!

<https://codeinplace.stanford.edu/cs49-w24/ide/a/headsup>





Heads Up!

- Read a list of CS-related words from a file into a list (this is provided as part of the starter code)
 - One person closes their eyes
 - The others display a random word from the list and describe the word
 - The code to display a random word from the list is our task
 - Our **random_color()** code for [random_circles](#) will come in handy
 - The person with their eyes closed guesses the word
 - When the guess is correct, a different person closes their eyes, and we rinse and repeat.
- 

The background is a solid pink color. It is decorated with various hand-drawn geometric shapes and lines in white and black. These include a dashed line in the top left, a white triangle in the top center, a black zigzag line in the top right, a white circle in the top right, two parallel black lines in the top right, a white triangle in the top right, a large black circle in the bottom right, a white circle in the bottom right, a black plus sign in the bottom left, a white circle in the bottom left, a white triangle in the bottom left, and a dashed line in the bottom left.

That's all, folks!

Next up: Dictionaries!