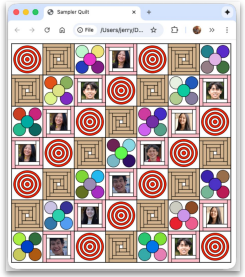
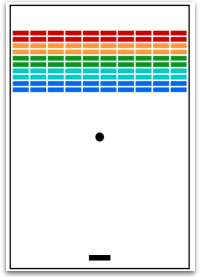
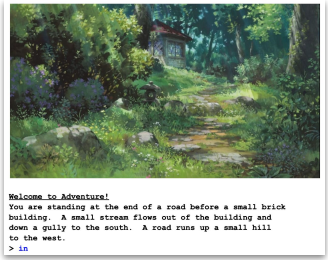
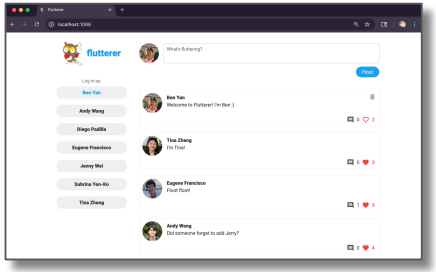









Note: The condensed slide deck!

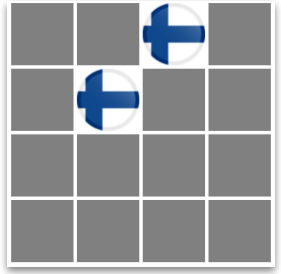
🎯 Final Review: Python, Client-Side JavaScript

CS106AX: Programming Methodologies in JS and Python

We're almost there, you got this!

Ben Yan, Dec 7th 2025

The big yellow flowers sigh warily tonight



CS106AX Autumn 2025

Stanford | ENGINEERING
Computer Science

Hello CS106AX! Happy Sunday!

Autumn

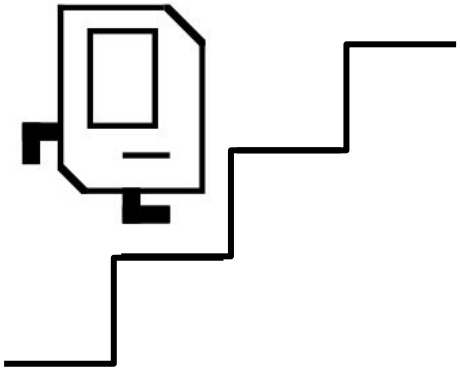
Winter

We're almost there – you're doing amazing!

Any fun winter break plans?

The Map For Today

1. Quick overview of final exam structure, logistics, what you can expect 📔
2. Recap of material: **Python, client-side JavaScript,** emphasis on idea over syntax
3. Walk through practice problems and coding exercises together 💖
4. Questions and office hours!



Final Logistics

🧭 Place & Time

- ❑ Time: **Monday**, December 8th, **8:30 AM - 11:30 AM**
- ❑ Location: **CoDa B90** (Lecture Room)
- ❑ For alternative arrangements, please reach out to Jerry asap!

✍️ Mechanics

- ❑ 3 hours long (+1 hour over midterm)
- ❑ 📚 **Open notes & book; closed computer**
- ❑ Focus on 🐍 Python and 📱 client-side JS (more on that on the next slide!)

The real CoDa to our journey was the friends we made along the way

or something like that

Final Exam Structure

- Identical to the Practice Final in structure and question categories!

Problem 1: Python Strings 🧵 (20 pts)

Problem 2: Python Strings and Lists 🔗 (20 pts)

Problem 3: Working with Python Dictionaries & Objects 📚 (20 pts)

Problem 4: Defining Python Classes & Reading Files 🗂️ (20 pts)

Problem 5: Client-Side JavaScript 📱 (20 pts)

Remember – this is all stuff you've worked with, and leveraged beautifully to make programs such as Adventure 🧗 and Flutterer 🦋. Keep doing what you do.

- **3 hour exam** – an approximate way to think about this is at least 30 min for each question, and extra 30 min for flexibility, though it really depends!

🎯 General Exam Tips and Recs

🧮 Broadly, problem-solving ability, logic, and **conceptual understanding** of the course material **is far more important than having perfect syntax**

- ❑ **We won't penalize for minor syntax errors, as long as your intentions are clear**, e.g., a missing brace in JS, or inconsistent spacing in Python *(as long as it's clear whether each line of code is under an if test, while loop, etc.)*
- ❑ Brief commenting is not required, but can help sometimes for partial credit!

⏱️ Try not to panic! *I know – way easier said than done.* But **you can do this!**

- ❑ Don't be afraid to move on + come back re-energized – and **try to attempt every problem**, even if you're not sure how to finish it off.
- ❑ Try not to rely too much on your notes / books / handouts, but it's great if you can **look up things, e.g., method names, effectively when necessary** 📗**.**

✍️ **Practice writing code on paper**, and go in with a strategy (e.g., does it help you to jot down some notes first, or even sketch your high-level approach?)

🎯 General Exam Tips and Recs

The question prompts can be quite long, information rich, and scary-looking!

➔ Personally, it helps me to first look at **smaller toy examples of input → output**

➔ Then more general examples, to identify a **general pattern or solution strategy.**

For longer solutions, I try breaking it down to **incremental (input → output) objectives**.

🗳️ **E.g., Ranked-choice voting problem** (midterm), **removing lowest candidate from ballots**

```
ballots = [["Jerry","Ben"], ["Jerry"], ["Jerry","Ben"], ["Ben"]]
```

📊 **Step 1:** Get **frequency map** (map[candidate] => numFirstVotes) from **list of ballots**

```
map = {"Jerry": 3, "Ben": 1}
```

😢 **Step 2:** Get **least popular candidate (string)** from **frequency map**

```
leastPopularCandidate = "Ben" :(
```

❌ **Step 3:** From **each ballot in list**, delete any appearance of **this candidate (string)**

```
ballots = [["Jerry",❌], ["Jerry"], ["Jerry",❌], [❌]]
```

🔢 Python: Range-Based Iteration

Unlike in JavaScript, where we iterates over number indices via

```
for (let i = start; i < limit; i++){ …
```

In Python Land, we generally use the built-in `range()` iterator function!

- ❏ `range(limit)` counts from `0` up to `limit - 1`
- ❏ `range(start, limit)` counts from `start` up to `limit – 1`
- ❏ `range(start, limit, step)` counts by `step`

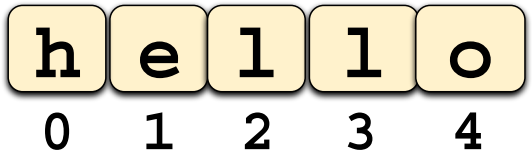
```
for i in range(3):
    print(i) #0 => 1 => 2
for i in range(2, 6):
    print(i) #2 => 3 => 4 => 5
for i in range(1, 100, 2):
    print(i) #1 => 3 => 5 => ... => 99
```

Note that we always stop one step before limit is reached!

Strings in Python

Sequence of characters (a single letter, number, symbol i.e. ASCII table) **in order**, enclosed by double quotes

`str = "hello"`



Strings are 0-indexed!

✍️ Methods: What we can do with a string?

`len(str)`

Returns length of string

`str[i]`

Returns character of string at **`index i`**

`substr = str[start:end]`

Extracts and returns substring between index start (inclusive) and end (exclusive), e.g., 🌎 `"earth"[0:3]` → `"ear"`

✅ *As a quick check, the substring should be (`end - start`) chars long*

`substr in str`

`true` if `str` includes `substr` as a substring (can be 1 char), else `false`

e.g., ✅ `"art" in "earth"`

`str.find(substr, start)`

Returns index in `str` where first instance of `substr` begins, or `-1` if not found.

`start:` Optional, index to start searching

Python Strings Pattern

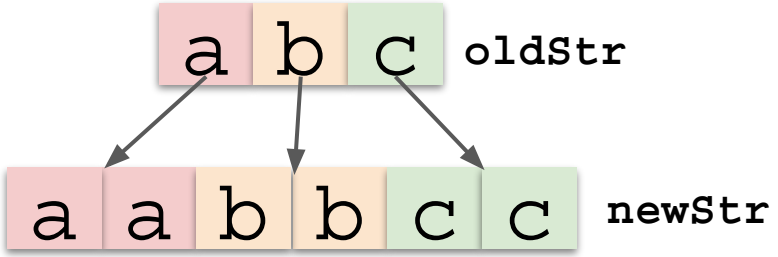
Helpful pattern with string functions: given a string, iterate through its characters and build up a **new string** (since strings are **immutable!** can't be changed*)

```
let oldStr = some string ...
let newStr = "";
for i in range(len(oldStr)):
    # build up newStr
```

```
# alternative syntax,
# directly over chars
for char of oldStr:
   # build up newStr
```

Example: `twins` function that takes an input string, and returns a copy of that string with each character repeated once, e.g., "`abc`"→"`aabbcc`"

```
def twins(str):
  result = ""
  for char in str:
      result += char + char
  return result
```



*String methods like concatenation create entirely new string objects that we assign to an existing/new variable!

String Methods

- Don't have to memorize them, but may be nice to have a reference sheet available!

Methods for Finding Patterns

str`.find(`*pattern*`)` Returns the first index of *pattern* in *str*, or –1 if it does not appear.
str`.find(`*pattern*`, `*k*`)` Same as the one-argument version but starts searching from index *k*.
str`.rfind(`*pattern*`)` Returns the last index of *pattern* in *str*, or –1 if it does not appear.
str`.rfind(`*pattern*`, `*k*`)` Same as the one-argument version but searches backward from index *k*.
str`.startswith(`*prefix*`)` Returns `True` if this string starts with *prefix*.
str`.endswith(`*suffix*`)` Returns `True` if this string ends with *suffix*.

Methods for Transforming Strings

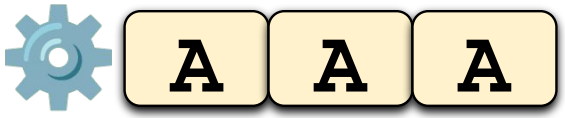
str`.lower()` Returns a copy of *str* with all letters converted to lowercase.
str`.upper()` Returns a copy of *str* with all letters converted to uppercase.
str`.capitalize()` Capitalizes the first character in *str* and converts the rest to lowercase.
str`.strip()` Removes whitespace characters from both ends of *str*.
str`.replace(`*old*`, `*new*`)` Returns a copy of *str* with all instances of *old* replaced by *new*.

Methods for Classifying Characters

ch`.isalpha()` Returns `True` if *ch* is a letter.
ch`.isdigit()` Returns `True` if *ch* is a digit.
ch`.isalnum()` Returns `True` if *ch* is a letter or a digit.
ch`.islower()` Returns `True` if *ch* is a lowercase letter.

ch`.isupper()` Returns `True` if *ch* is an uppercase letter.
ch`.isspace()` Returns `True` if *ch* is a ***whitespace character*** (space, tab, or newline).
str`.isidentifier()` Returns `True` if this string is a legal Python identifier.

Python Strings Example!



Recall from the A4: Enigma that **rotor setting strings**, e.g., **"AAA", "BZZ"**, are used to represent the positions of the 3 rotors. When clicking a key, the rotors advance as a base-26 car odometer might, e.g., **"JLY" becomes "JLZ"**, and with a second click, **"JLZ" advances to "JMA"** – the "Z" wraps around to an A", triggering the next rotor to advance.

However, this idea **doesn't have to be confined to just rotor settings of length 3.**

```
advanceRotorSetting("JLY") => "JLZ"
advanceRotorSetting("BZZ") => "CAA"
advanceRotorSetting("ZZZ") => "AAA"
advanceRotorSetting("BEESBUZZ") => "BEESBVAA"
advanceRotorSetting("QZZZZZZZZZ") => "RAAAAAAAAA"
```

note the double carry!

Everything rotates back to start

note we keep triggering the next rotor to advance until a rotor doesn't wrap around to "A"

Task: Write a function that takes in an **uppercase rotor string (any length)**, and **synthesizes a new rotor string** that would result from a single key click event i.e. **advanced by one.**

advanceRotorSetting

Task: Write a function that takes in an **uppercase rotor string (any length)**, and **synthesizes a new rotor string** that would result from a single key click event i.e. **advanced by one.**

❓ For this problem, questions I might ask myself:

➔ **How should I iterate through the original string? Forward or in reverse?**

Probably **reverse, as the rotors start advancing from the end of the string** (and that way, we know if we should carry over to the neighboring rotor to the left)

➔ **What do I do with each character / rotor of the original string?**

If we haven't stopped advancing yet, we should advance it, e.g., **A→B, Z→A**

➔ **How do I check if I should keep advancing?**

Keep a state boolean variable, e.g., **`keepAdvancing`** or **`carry`**, that starts as **`True`**, flipping to **`False`** only when a rotor doesn't wrap around back to A.

➔ **Is there anything else I need to think about? Any special cases?**

One thing, handling the Z→A wrap-around, and not trying to get a letter after Z lol ❌
Also, due to reverse iteration, when adding rotor letters to the new string, add to the front

advanceRotorSetting

Task: Write a function that takes in an **uppercase rotor string (any length)**, and **synthesizes a new rotor string** that would result from a single key click event i.e. **advanced by one.**

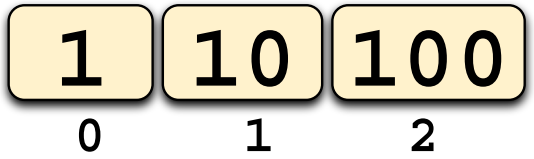
Solution ✅

```
ALPHABET = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
def advanceRotorSetting(setting):
    nextSetting = ""
    keepAdvancing = True
    for i in range(len(setting) - 1, -1, -1): #reverse iteration
        rotorLetter = setting[i]
        if (keepAdvancing):
            letterIndex = ALPHABET.find(rotorLetter)
            newLetter = ALPHABET[(letterIndex + 1) % 26]
            keepAdvancing = newLetter == 'A' #check if wrap-around
        else:
            newLetter = rotorLetter #don't advance rotor, keep same
        nextSetting = newLetter + nextSetting #add to front
    return nextSetting
```

Any questions on strings?

Lists in Python

```
arr = [1,10,100]
```



Sequence of elements of any type, with indices starting at `0` and ending at `len(arr) - 1`, akin to strings.

`len(arr)` `3` `arr[0]` `1` `arr[len(arr)-1]` `100` `arr[0:2]` `[1,10]`

Two crucial techniques / programming idioms with an array include:

1. Iterating through its elements in forward order
2. And in reverse order 🙃

1) Forward Iteration ➡️

```
# could also write for element of arr:
for i in range(len(arr)):
    element = arr[i]
    # code to process each element in turn
```

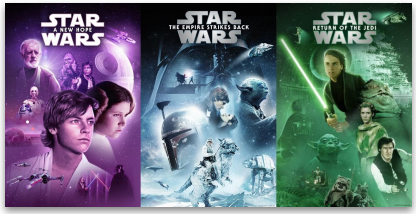
2) Reverse Iteration ⬅️

```
for i in range(len(arr)):
    element = arr[len(arr)-1-i]
    # code to process each element
    # in turn, starting from the end
```

If forward doesn't seem to work well for a given task, give reverse a try! ⏪

List Methods

- Don't have to memorize them, but may be nice to have a reference sheet!



Methods that Return Info

list`.index(`*value*`)` Returns the first index at which *value* appears in *list* or raises an error.
list`.index(`*value*`, `*start*`)` Returns the first index of *value* after the starting position.
list`.count(`*value*`)` Returns the number of times *value* appears in *list*.
list`.copy()` Creates a new list whose elements are the same as the original.

Methods that reorder elements

list`.reverse()` Reverses the order of elements in the list.
list`.sort()` Sorts the elements of *list* in increasing order.
list`.sort(`*key*`)` Sorts the elements of *list* using *key* to generate the key value.
list`.sort(`*key*`, `*reverse*`)` Sorts in descending order if *reverse* is `True`.

Methods that Add/Remove Elements

list`.append(`*value*`)` Adds *value* to the end of the list.
list`.insert(`*index*`, `*value*`)` Inserts *value* at the specified index, shifting subsequent elements over.
list`.remove(`*value*`)` Removes the first instance of *value*, or raises an error if it's not there.
list`.pop()` Removes and returns the last element of the list.
list`.pop(`*index*`)` Removes and returns the element at the specified index.
list`.clear()` Removes all elements from the list.

Methods that involve strings

str`.split()` Splits a string into a list of its components using whitespace as separator.
str`.split(`*sep*`)` Splits a string into a list using the specified separator.
str`.splitlines()` Splits a string into separate lines at instances of the newline character.
sep`.join(`*list*`)` Joins the elements of *list* into a string, using *sep* as the separator.

Slicing

```
arr[start:finish]
```

Similar to substrings: returns subarray from index ***start*** to ***finish-1***, inclusive.

Python Lists Example! Insertion Sort

Task: Implement a function **insort(words, newWord)** that takes as input:

➔ **words:** An alphabetically sorted (by increasing order) list of strings

➔ **newWord:** A string that is not currently in **words**

and returns a copy of the list with the new word added, and the alphabetic sorting order maintained.

Some example calls to **`insort`** include:

- **insort**([”fire”, ”water”], ”air”) **=>** [“air”, “fire”, ”water”]
- **insort**([“air”,”fire”, ”water”], ”earth”) **=>** [“air”,”earth”, “fire”, ”water”]
- **insort**([“blue”, “green”] , “red”) **=>** [“blue”, “green”, “red”]
- **insort**([] , “banana”) **=>** [“banana”]

Note that you can use **`str1 < str2`** to compare strings alphabetically in Python!

Python Lists Example! Insertion Sort

Task: Implement a function **insort(words, newWord)** that returns a new copy of the **words** list with the **newWord** added and increasing alphabetic order maintained.

❓ For this problem, questions I might ask myself:

➔ **What do I do while scanning through each index i of the original list?**

See if I should insert the new word at that index

➔ **What needs to happen if I want to insert the new word at index i?**

For the original element at index i, and all elements onwards, I need to shift them down

➔ **At a given index i, how would I know to insert the new word there?**

If alphabetically, **words[i–1] < newWord < words[i]**. But from the previous loop iteration, we would have already checked with words[i-1], so we can simply check if **newWord < words[i] !**

➔ **Is there anything else I need to think about? Any special cases?**

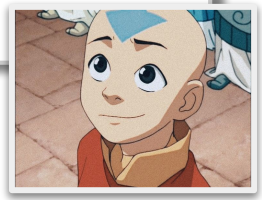
Inserting at the **very end of the list**, if no current elements are greater than newWord

Python Lists Example! Insertion Sort

Task: Implement a function **insort(words, newWord)** that returns a new copy of the **words** list with the **newWord** added and increasing alphabetic order maintained.

```
def insort(words, newWord):
    for i in range(0, len(words)): # index to insert
        if (newWord < words[i]):
            # insert the new word at index i
            words = words[:i] + [newWord] + words[i:]
            # return modified list
            return words

    # if no current elements larger than newWord, insert at end
    words = words + [newWord]
    return words
```

Solution ✅



Any questions on lists?

Dictionaries in Python

JavaScript Objects and it's the same except there's new quirks so it's not

🔑 You define a dictionary/map by **listing a collection of key-value pairs**. Keys are unique; values are not necessarily unique. **Keys must be of immutable type, e.g., strings, ints, tuples**

```
bratAlbum = {
  "artist": "Charli XCX",
  "streams": 1000,
   1: "360", # title song
   2: "Club Classics"
} // syntax is key:value,
```

Can iterate through dictionary as follows:

```
for key in map:
   value = map[key]
   // code to work with individual key & value
```

As of Python 3.7, they are insertion-ordered!

🔑 We can **select, add, or modify a key-value pair** via an expression of the form: `map[key]`

⚠️ You may want to **make sure to check** `if key in map` **first if needed**

bratAlbum["streams"] → 1000

bratAlbum["streams"] *= 4

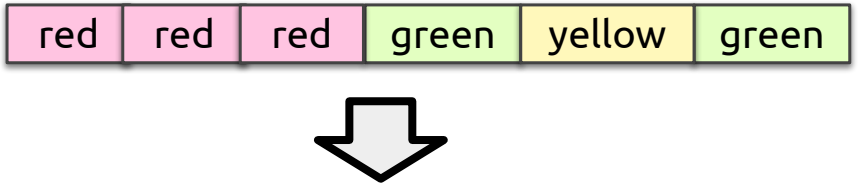
4000

🔑 streams

Some neat dictionary-related Python tasks

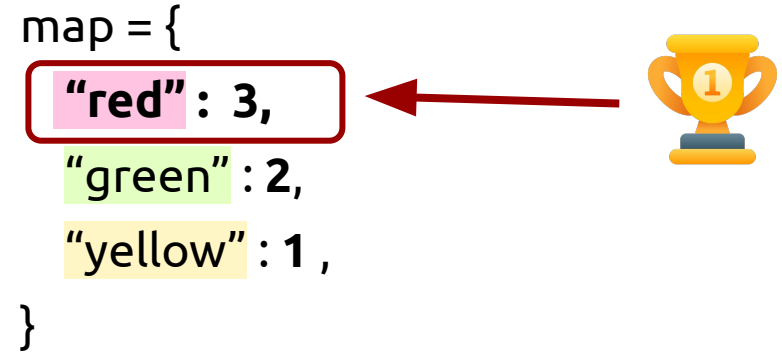
Building up a frequency map / count dictionary of elements in a list

```
frequencyMap = {}
for element in someList:
    if element in frequencyMap:
        frequencyMap[element] += 1
    else: # first time seeing element
        frequencyMap[element] = 1
```



For a map / dictionary, finding the key affiliated with the largest value

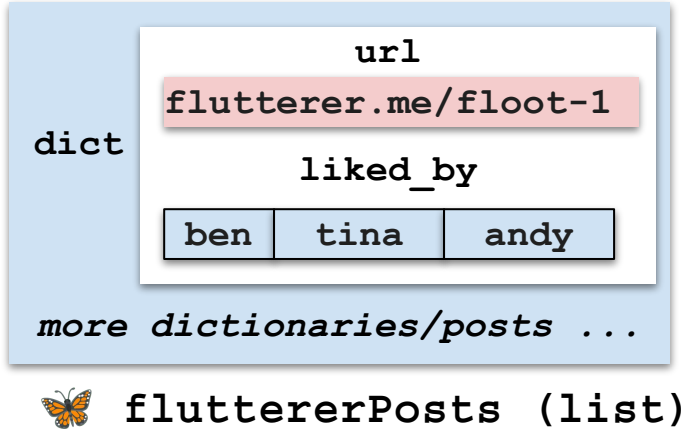
```
currBestKey = None
currBestValue = -1 # or min possible
for key in map:
    value = map[key]
    # see if we beat the current best
    if (value > currBestValue):
        currBestKey = key
        currBestValue = value
```



Working with Nested Data Structures

Nested data structures, like lists of lists, list of dictionaries, dictionaries with lists or even other dictionaries inside as values, oh my, **can be tough!** 😨

```
fluttererPosts = [
   {"url": "flutterer.me/floot-1",
    "liked_by": ["ben","tina","andy"]},

   {"url": "flutterer.me/floot-2",
    "liked_by": ["eugene","jenny"]},

   ... more flutterer posts
]
```



fluttererPosts (list)

- **Advice:** Think one level of nesting at a time, and at each level (e.g., for an array of objects, `list` → `dictionary` → `key,value` **pair within** `dictionary`), it's helpful to know what kind of data type you're working with.
- **After all, an** `list` **of lists is still a** `list` **at the end of the day,** as is an `list` of `dictionaries`, and can be treated as such (e.g., can still loop over elements)

Working with Nested Data Structures

Advice: You may even **name your variables** to make it clear what each nested one is!

Printing out the cells of a Tic-Tac-Toe board!

```
ticTacToeBoard = [["X"," "," "],
                  ["O","X"," "],
                  ["O","X","O"]]
```

```
for row in ticTacToeBoard:
    for cell in row:
        print(cell)
```

Printing out which user liked what Floot post in Flutterer!

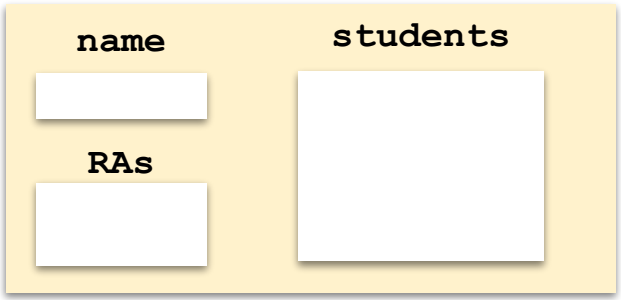
```
flutterer Posts = [
    {"url": "flutterer.me/floot-1",
     "liked_by": ["ben","tina","andy"]},

    {"url": "flutterer.me/floot-2",
     "liked_by": ["eugene","jenny"]},

    ... more flutterer posts
]
```

```
#post is dict, keys="url","liked_by"
for post in flutterer Posts:
    url = post["url"] #string
    liked_by = post["liked_by"]
    #loop through list (of strings)
    for user in liked_by:
        #user is string
        print(user + "liked " + url)
```

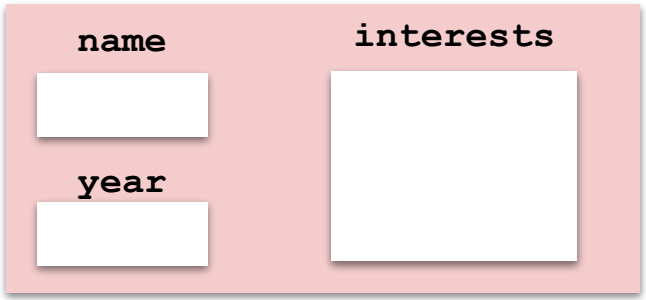
Python Classes as Templates

- Defines a **new data type** that our program can use.
  - A ***class*** is a blueprint that defines the structure and behavior of a certain data type.
  - An ***object*** is a value that belongs to a class.

- A single class (e.g., ***Dorm***) can fashion any number of objects – each of which is an **instance** of that class (e.g., ***Burbank, Larkin, Crothers***).



Dorm



Student

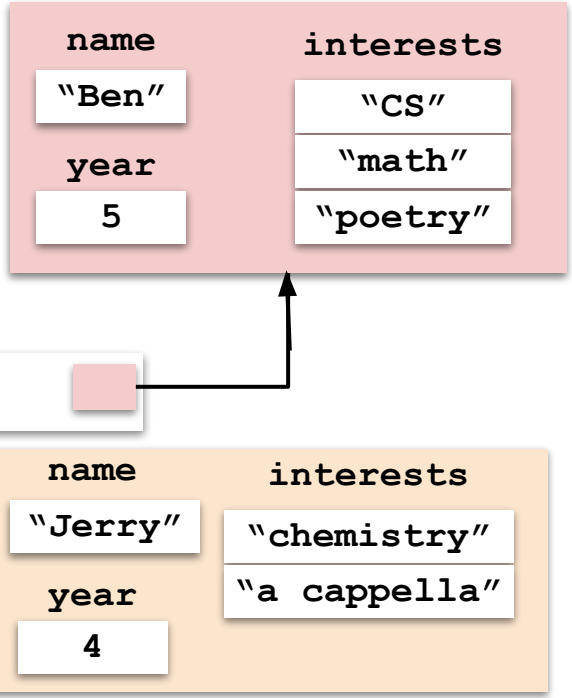
Python Classes and Objects

- An example ***class*** for a new **`Student`** data type—a template from which we create individual **`Student`** data ***objects***.

```
class Student:
    #constructor method (initializes attributes)
    def __init__(self, name, year, interests):
        self.name = name
        self.year = year
        self.interests = interests
    #add more methods here ...
```

```
ben = Student("Ben", 5, ["CS","math","poetry"])
```

```
jerry = Student("Jerry", 4,
["chemistry","a cappella"])
```



Python Classes as Templates

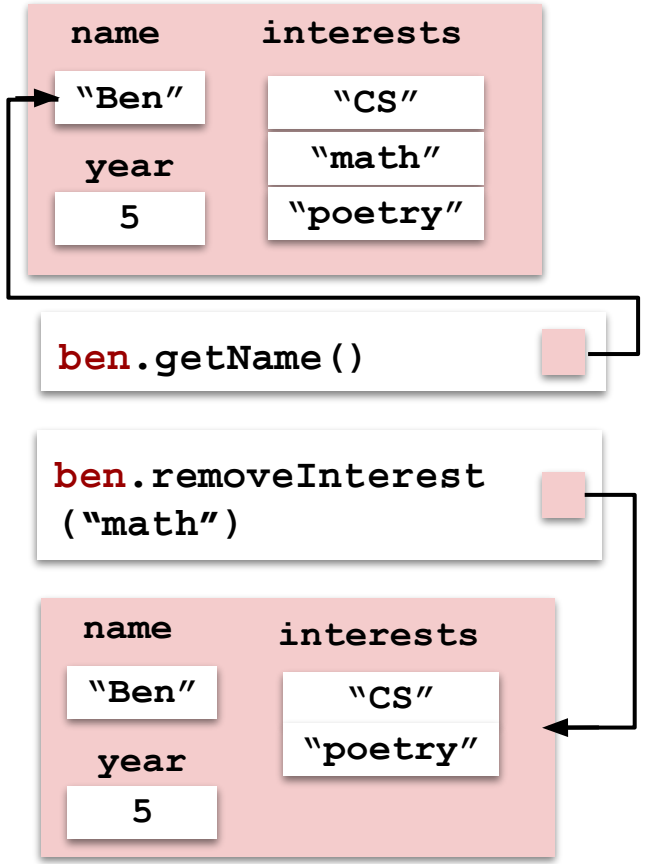
poetry >> JavaScript

- Classes typically include ***getter*** and ***setter*** methods, for ***accessing/returning*** and ***modifying*** the internal attributes, respectively.

```
def getName(self): #getter
    return self.name

def setYear(self, newYear): #setter
    self.year = newYear

def removeInterest(self, interest):#setter
    if interest in self.interests:
        self.interests.remove(interest)
```

General syntax for calling methods

```
objectName.methodName(parameters)
```



File Reading

- In data-driven programs, **we often store data** (e.g., Adventure rooms, objects) **in files, which we then read in Python**, organizing the parsed content in data objects.
- To iterate over the (identically formatted) lines in a file, we can write:

airports.txt

```
SFO: San Francisco
LAX: Los Angeles
MSP: Minneapolis
PEK: Beijing
XYZZY: AdventureLand
... more airports
```

```
airportMap = {}
with open("airports.txt") as f:
    for line in f:#each iteration is 1 airport
        sep = line.find(":")
        code = line[:sep]
        location = line[sep + 1:]
        airportMap[code] = location
```

File Reading

- In data-driven programs, **we often store data** (e.g., Adventure rooms, objects) **in files, which we then read in Python**, organizing the parsed content in data objects.
- To iterate over the (identically formatted) lines in a file, we can write:

```
airports.txt
SFO: San Francisco
LAX: Los Angeles
MSP: Minneapolis
PEK: Beijing
XYZZY: AdventureLand
... more airports
```

```
airportMap = {}
with open("airports.txt") as f:
    for line in f:#each iteration is 1 airport
        sep = line.find(":")
        code = line[:sep].strip()
        location = line[sep + 1:].strip()
        airportMap[code] = location
```

- In cases where each line is structured differently, but the file can still be neatly delineated into similar sections (e.g., Adventure rooms, CFG grammar files), **`while True` can be very useful.** We'll work with an example of this next.

Python: File Reading Exercise (Dorms!!)

🏘️ Python: File Reading Exercise

- Suppose that we, 🌲 a Stanford RA, have **a roster of students in a dorm, stored inside a text file**, listing each student's name, year, and interests!
- Building off the `Student` data class, we want to build a 🏘️ `Dorm` class—with internal attributes
  - **self.students** as a dictionary mapping from all Student names to `Student` data objects
  - **self.sharedInterests** as a dictionary mapping from each interest to a list of student names who possess that interest, e.g.,

studentRoster.txt

```
Ben Ten
5
CS,boba,poetry,K-pop

Eugene Genius
3
boba,art,tea,math,CS

Jerry Terrier
4
CS,art,poetry,choir
...
```

```
self.sharedInterests["boba"] => ["Ben Ten", "Eugene Genius", ...]
```

- This data structure can be useful for planning events! (like boba parties 🧋)

Python: File Reading Exercise

Task: Write a constructor method with an input **`filename`**. The method should parse the file (student roster) and populates the attributes **self.students** and **self.sharedInterests**. For this task, you can guarantee that names are unique, and that one line separates each student entry.

studentRoster.txt

```
Ben Ten
5
CS,boba,poetry,K-pop

Eugene Genius
3
boba,art,tea,math,CS

Jerry Terrier
4
CS,art,poetry,choir
...
```

Solution ✅

```
def __init__(self, filename):
    self.students = {}
    self.sharedInterests = {}
    with open(filename) as f:
        while True: #each iteration corresponds to one student
            name = f.readline().strip()
            if name == "": break #no more entries in file
            year = int(f.readline().strip())
            interests = f.readline().strip().split(",")
            f.readline() #skip the in-between line

            #construct Student data object from attributes
            self.students[name] = Student(name, year, interests)

            #populate sharedInterests dictionary
            for interest in interests:
                if interest not in self.sharedInterests:
                    self.sharedInterests[interest] = []
                self.sharedInterests[interest].append(name)
```

Python: File Reading Exercise

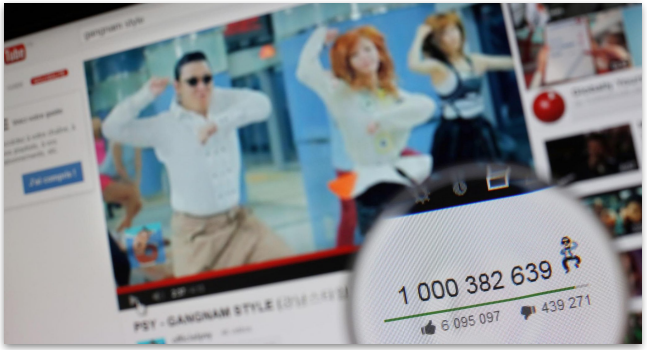
Task: Write a constructor method with an input **`filename`**. The method should parse the file (student roster) and populates the attributes **self.students** and **self.sharedInterests**. For this task, you can guarantee that names are unique, and that one line separates each student entry.

studentRoster.txt

```
Ben Ten
5
CS,boba,poetry,K-pop

Eugene Genius
3
boba,art,tea,math,CS

Jerry Terrier
4
CS,art,poetry,choir
...
```

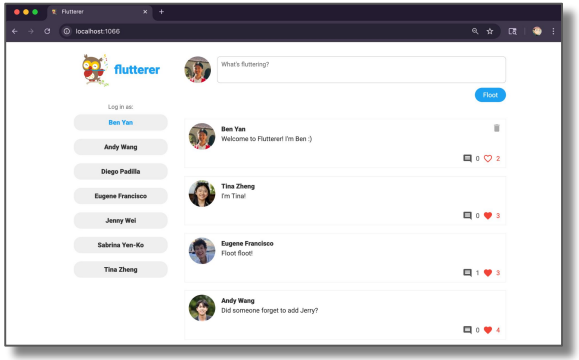
Solution ✅

```
def __init__(self, filename):
    self.students = {}
    self.sharedInterests = {}
    with open(filename) as f:
        while True: #each iteration corresponds to one student
            name = f.readline().strip()
            if name == "": break #no more entries in file
            year = int(f.readline().strip())
            interests = f.readline().strip().split(",")
            f.readline() #skip the in-between line
            #construct Student data object from attributes
            self.students[name] = Student(name, year, interests)
            #populate sharedInterests dictionary
            for interest in interests:
                if interest not in self.sharedInterests:
                    self.sharedInterests[interest] = []
                self.sharedInterests[interest].append(name)
```



Web Programming!

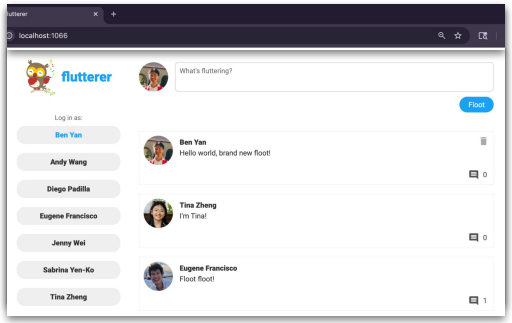
Fun Fact: Every 🌲 student has a website at stanford.edu/~[SUNET]



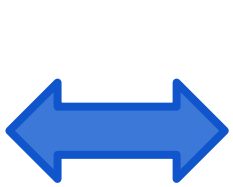
Client-Server Model

The 📱 **client side**—browser webpage and supporting JavaScript code—becomes truly interactive for the user when **communicating with an active Python server.**

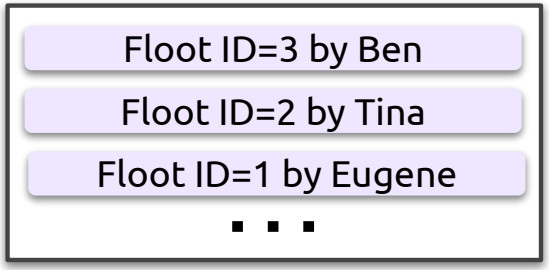
➢ Client can, e.g., fetch resources from the server, or update the server data.



Web Client (Browser)



Server

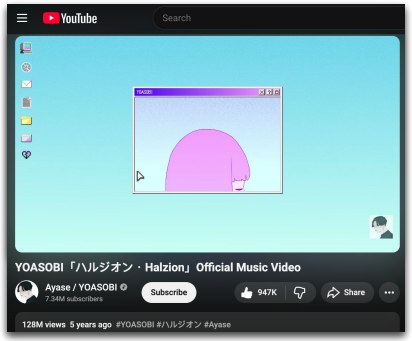


Server Database

❑ **Please note:** Server-side Python is not tested on the final! **What is tested is how the JavaScript client sends and receives / handles `AsyncRequests`.**

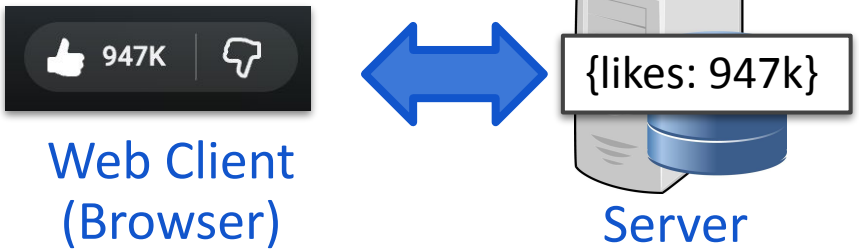
Building Web Applications – AsyncRequest

- **Thin-Client:** Most web apps minimize the amount of data needed for initial user experience and interaction – e.g., an initial HTML outline / skeleton
- **As new resources are needed** (e.g., images to load, videos, new content), **the website can download them from server in the background i.e. *asynchronously.***
- Once downloaded, JavaScript can update the DOM tree to include those resources on the webpage.



Key Idea: We need to **make sure the client and server as on the same page (get it?)**

- The browser's presentation of data / information **should be in sync** with the information stored in the server.

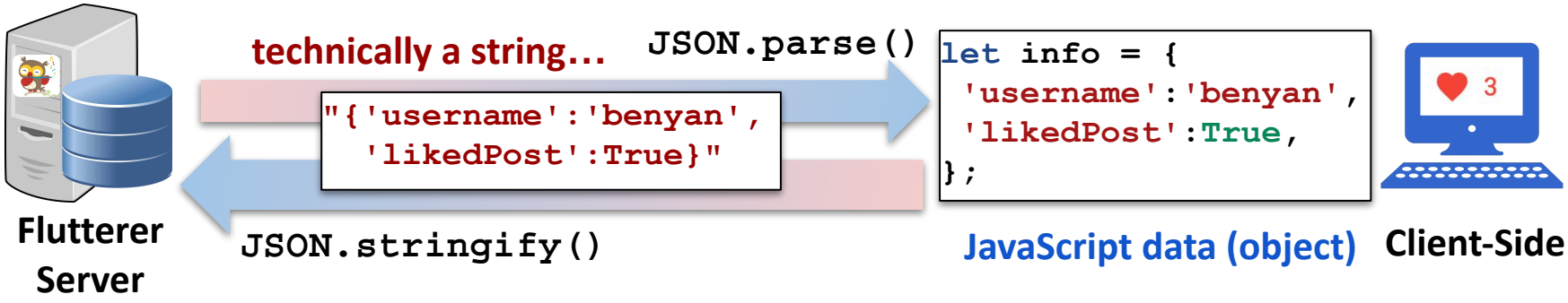


Building Web Applications – AsyncRequest

- **AsyncRequests** are framed as 🔗 URLs / API call strings, such as

```
AsyncRequest("graph.facebook.com/me")
AsyncRequest("numbersapi.com/122")
AsyncRequest("numbersapi.com/122?json=true")
AsyncRequest("/api/floots")
```

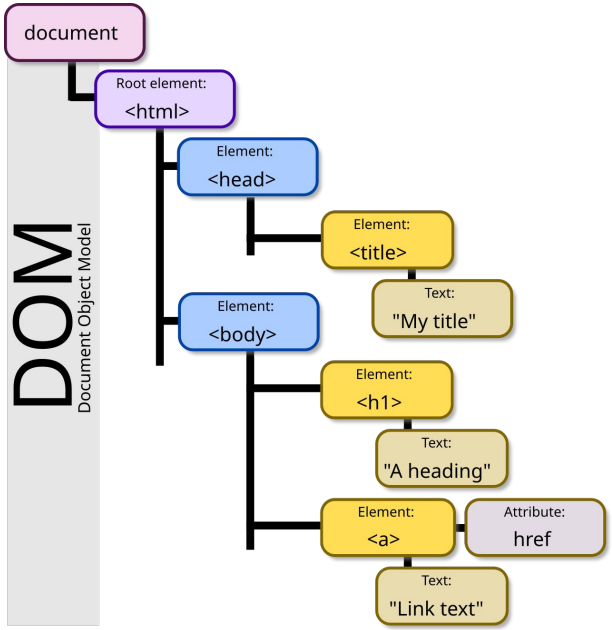
- **AsyncRequests** generally fetch data from the server as plaintext JSON (block of text / string form) – which is incorporated via JavaScript into an HTML element.



DOM Tree

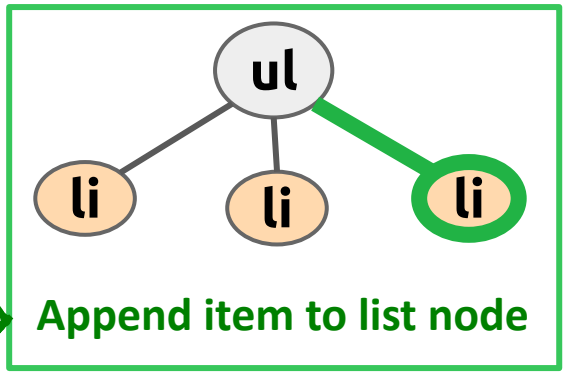
- The browser **translates an HTML document** into an **internal Document Object Model (DOM) tree**, which can interface with JavaScript programs.
- DOM is hierarchy of data objects called ***elements***, usually a paired set of tags.

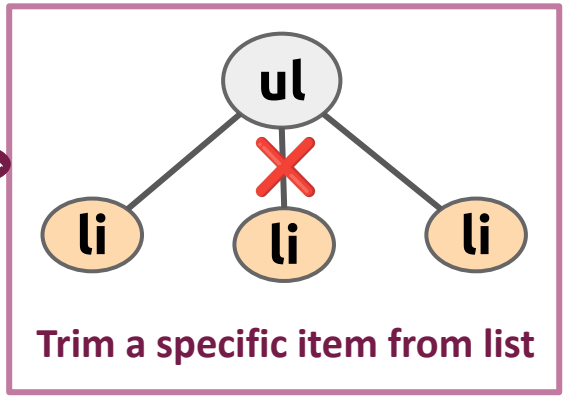
```
<!DOCTYPE html>
<html>
  <head>
    <title>My title</title>
  </head>
  <body>
    <h1>A heading</h1>
    <a href=”A LINK”>Link
text</a>
  </body>
 </html>
```



Interacting with the DOM Tree in JavaScript

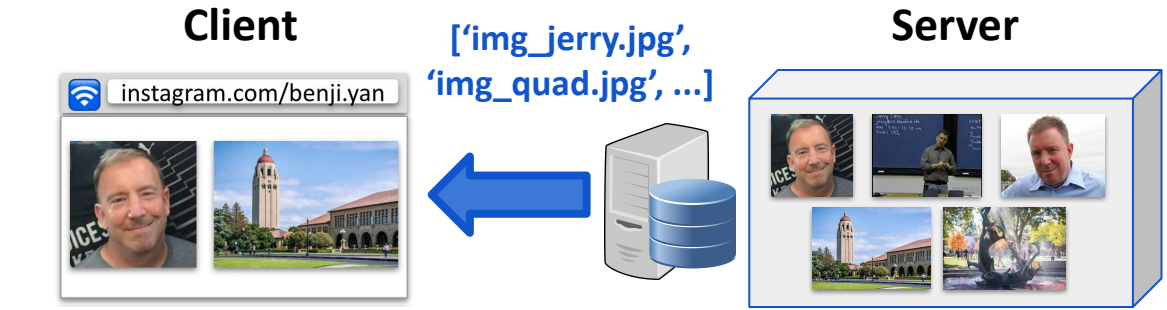
`document.getElementById(`*id*`)`	Returns the element with the specified id attribute.
`document.createElement(`*type*`)`	Creates a new tag element for the tag of the specified type.
`document.createTextNode(`*text*`)`	Creates a new text element for the provided text string.
element`.getElementsByTagName(`*name*`)`	Returns an array of the elements with the specified tag name.
element`.appendChild(`*node*`)`	Appends a node to the end of the element's list of children.
element`.removeChild(`*node*`)`	Removes a node from the element's list of children.
element`.parentNode` *element*`.lastChild`	Stores references to a node's parent and last child.





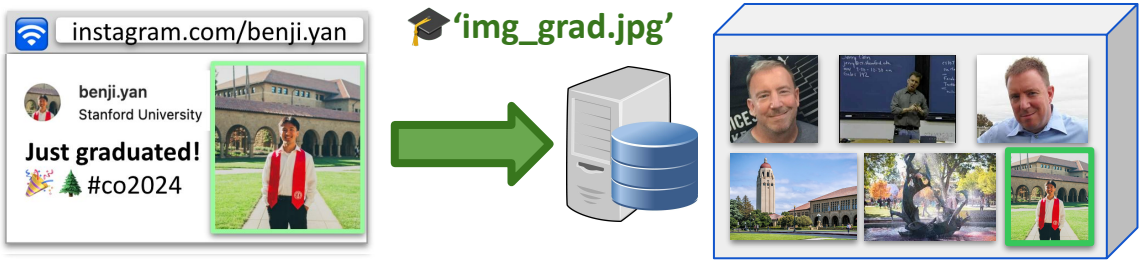
📫 Client-Side AsyncRequests: GET vs POST

📘 For a **GET**, typically you **ask the database (without modifying)** for requested info, which it sends back to you (client)



e.g., when you load your IG feed
(payload is server → client)

✍️ For a **POST**, typically you (the client) **send info to the server**, e.g., a new posted photo, and request to **modify the database**



e.g., when you post a new photo
(payload is client → server)

JSON.parse(), JSON.stringify()

What exactly does "json stringify" mean? Are we reading Shakespeare?

When payload goes from …

Server to Client e.g., on client side, payload received by **GET** request	**`JSON.parse(payload)`** Turn a **payload string** from the server into the corresponding **data structure** in JavaScript, like **literally removing the quotation marks.**
Client to Server e.g., on client side, payload sent by **POST** request	**`JSON.stringify(data)`** Does the reverse, transforming a JS **data structure** (e.g., array, object) into a **string** version, like **putting "" marks around it.**

Flutterer Server

technically a string…

`"{'username':'benyan', 'likedPost':True}"`

`JSON.parse()`

`JSON.stringify()`

```
let info = {
 'username':'benyan',
 'likedPost':True,
};
```

JavaScript data (object)

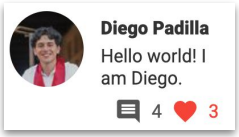
Client-Side

AsyncRequest: GET

- A GET asynchronous request has the **general template:**

```
let req = AsyncRequest( Write name of API Endpoint );
req.setMethod("GET"); //removing this line OK, default is "GET"
req.setSuccessHandler(function(response) {
    let payload = response.getPayload(); // a block of text
    let info = JSON.parse(payload); // turned into JS data type
    Write code that runs when info is successfully fetched from server
});
req.send(); //sends the AsyncRequest from client to server
```

- **Note:** `info` is a **JS data structure** with data fetched from the server
- For instance, an **array** of image filenames (strings), an **array** of floot objects, or a single **JavaScript object** corresponding to a specific floot



AsyncRequest: GET (Version II)

- You can also 🔗 daisy-chain the **`AsyncRequest`** calls, and define the success handler function separately (passing its name into **`setSuccessHandler`**), like this:

```
function successHandler(response){
    let payload = response.getPayload(); //JSON string
    let info = JSON.parse(payload); //turned into JS data type
    Write code that runs when info is successfully fetched from server
}
let req = AsyncRequest( Write name of API Endpoint )
            .setMethod("GET") //redundant, but for emphasis
            .setSuccessHandler(successHandler)
            .send();
```

- **Both styles are hot stuff**—up to you which one you're more comfortable writing, or which one feels more convenient for the problem! I personally like this one.

AsyncRequest: Inventory of Calls

`AsyncRequest``(`*url*`)`	Creates a request object primed to fetch a document from *url*.
request`.`**`addParam`**`(`*key, value*`)`	Adds a key-value pair to the *url*'s of query string.
request`.`**`addParams`**`(`*params*`)`	Adds all key-values pairs in *params* to the *url*'s query string.
request`.`**`setSuccessHandler`**`(`*callback*`)`	Sets the *callback* to be executed when the server responds.
request`.`**`setErrorHandler`**`(`*callback*`)`	Sets the *callback* to be executed if the server fails to fetch the *url*.
request`.`**`send`**`()`	Initiates the request for the relevant resource.
response`.`**`getPayload`**`()`	Returns the payload from the *response* passed to your *callback*.

Note: CS106AX uses a custom, in-house **`AsyncRequest`** class, as JavaScript's built-in asynchronous libraries use JS features beyond the course's scope.

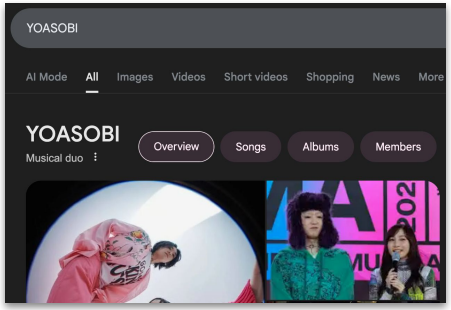
AsyncRequest: Parameters

- **Can add specific parameters** (*key:value* pairs) to your AsyncRequest to server!

Want to use the Google Search API to look up YOASOBI?

```
GET www.google.com/search?q=YOASOBI
```

```
let req = AsyncRequest("www.google.com/search")
            .addParam('q','YOASOBI')
            ... other daisy-chained calls
            .send();
```

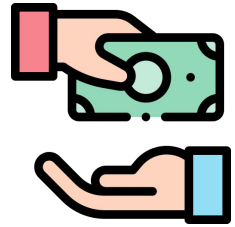


Want to look up YOASOBI, but now with a host language of Japanese? (hl = ja)

```
GET www.google.com/search?q=YOASOBI&hl=ja
```

```
let req = AsyncRequest("www.google.com/search")
            .addParams({'q':'YOASOBI','hl':'ja'})
            ... other daisy-chained calls
            .send();
```

AsyncRequest: POST



- A POST asynchronous request has the **general template:**

```
let req = AsyncRequest(
```

Write name of API Endpoint

```
);
req.setMethod("POST"); //line is needed; default is "GET"
req.setPayload(JSON.stringify(
```

Info / data structure to send over to the server / database

e.g., array of new users, [“Kieran”, “Linda”, “Isaias”] 🪪

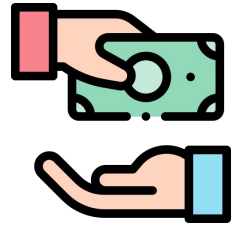
```
));
req.setSuccessHandler(function(response) {
```

Write code that runs when info is successfully handled by server

e.g., might add the new Flutterer users to the sidebar 👤

```
});
req.send();
```

AsyncRequest: POST (Version II)



- Like a GET request, can also write it like this instead – up to you! :)

```
let info =
```

JavaScript data structure that needs to be sent to server

Can be really any type but array/object is most likely

```
function successHandler(response){
```

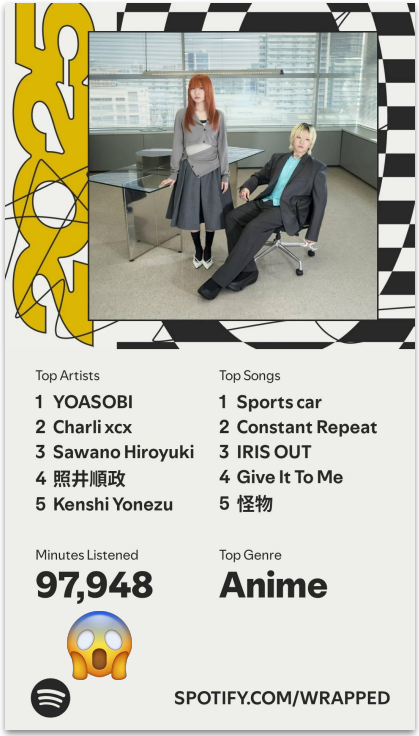
Write code that runs when info is successfully handled by server

```
}

let req = AsyncRequest(
```

Write name of API Endpoint

```
)
    .setMethod("POST") //line is needed; default is "GET"
    .setPayload(JSON.stringify(info)) // payload is texting :)
    .setSuccessHandler(successHandler)
    .send();
```

AsyncRequest Example: 🎁 Spotify Wrapped!

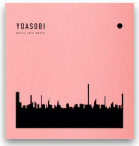


I do love JJK though 

AsyncRequest Example: 🎁 Spotify Wrapped!

Task: To populate a **<ol>** ordered list (**id="topAlbums"**) with images of our top 5 albums! 🎶

1. 

2. 

3. 
4. 
5. 

Say we have a GET API endpoint 🌐**/notSpotify/topAlbums?user=sunet**, and upon being invoked successfully, returns a JSON string structured as:

`"['Brat','TheBook','Sour','TheAlbum','Lover","1989",...]"`

List of at least 5 album names from most listened-to to least

Upon receiving this ranked list of album names from the server, we should make requests to 🌐**/notSpotify/images?album=album-name**, another GET endpoint which will fetch the image url for one album, e.g.,

`{"url": "images/CharliXCX-Brat.jpg", "found": True}`

Our approach is as follows: we make a request for the top album image first; when the server responds, we attach a **<li>** child node to the list, **and if found, and an <img> child to the <li> list item**. Only then so we issue a request for the next (2nd) album image, and repeat the same process, **up to five albums.**

Note that we have at most 1 AsyncRequest active at a time! (relay race)

AsyncRequest Example: 🎁 Spotify Wrapped!

Task: To populate a **<ol>** ordered list (**id="topAlbums"**) with images of our top 5 albums! 🎶

1. 

2. 

3. 
4. 
5. 

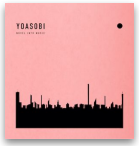
Solution ✅

```
function listTopAlbums(sunet, maxAlbums = 5){
  let HTMLList = document.getElementById("topAlbums");
  let albums = []; // populated by AsyncRequest to /topAlbums
  let index = 0; // album[index] is current album whose image to download
  let req = AsyncRequest("notSpotify/topAlbums?user=" + sunet);
  req.setSuccessHandler(function(response){
     albums = JSON.parse(response.payload()); // sets ranked albums list
     downloadNextAlbumImage(); // starts downloading first one
  });
  req.send();
  function downloadNextAlbumImage(){
     let imageReq = AsyncRequest("/notSpotify/images");
     imageReq.setParam("album", albums[index]);
     imageReq.setSuccessHandler( function(response){
         addAlbumImage(JSON.parse(response.payload()));
         if (++index === maxAlbums) return; // increment index, stop at 5
         downloadNextAlbumImage(); // start downloading next album image
     );
     imageReq.send();
  }
  function addAlbumImage(albumInfo){ // albumInfo => {"url":...,"found":...}
     /* assumes this works in adding the album to the HTML list */
  }
}
```

Optional: addAlbumImage

🎁 Spotify Wrapped!

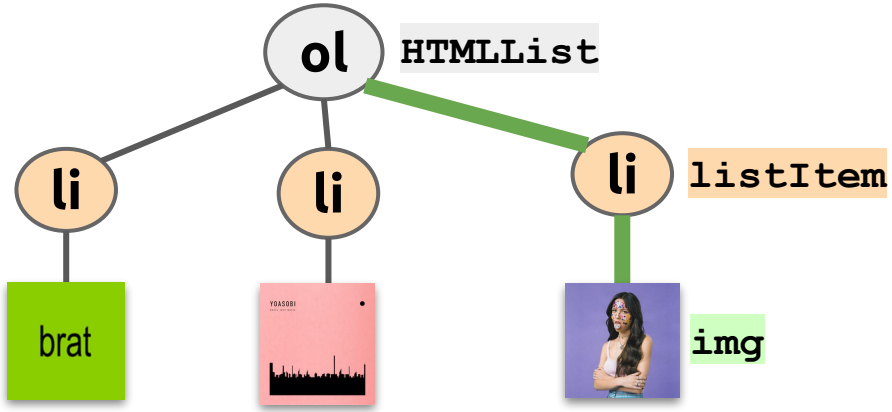
Task: To populate a **<ol>** ordered list (**id="topAlbums"**) with images of our top 5 albums! 🎶

1. brat
2. 
3. 

```
/* adds the album as a new item to the HTML list */
function addAlbumImage(albumInfo){
    let listItem = document.createElement("li");
    HTMLList.appendChild(listItem);
    // if image not found, skip adding image node
    if (!albumInfo["found"]) return;
    let img = document.createElement("img");
    img.setAttribute("src", albumInfo["url"]);
    listItem.appendChild(img);
}
```

Helper function solution ✅

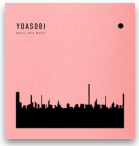
3. 



AsyncRequest Example: 🎁 Spotify Wrapped!

Task: To populate a **<ol>** ordered list (**id="topAlbums"**) with images of our top 5 albums! 🎶

1. 

2. 

3. 
4. 
5. 

Full solution ✅

```
function listTopAlbums(sunet, maxAlbums = 5){
  let HTMLList = document.getElementById("topAlbums");
  let albums = []; // populated by AsyncRequest to /topAlbums
  let index = 0; // album[index] is the current album whose image to download
  let req = AsyncRequest("notSpotify/topAlbums?user=" + sunet);
  req.setSuccessHandler(function(response){ // sets ranked list of albums, starts downloading first
     albums = JSON.parse(response.payload());
     downloadNextAlbumImage();
  });
  req.send();
  function downloadNextAlbumImage(){ // issue AsyncRequest for album[index], the next one up
       let imageReq = AsyncRequest("/notSpotify/images?album=" + albums[index]);
       imageReq.setSuccessHandler( function(response){
           let albumInfo = JSON.parse(response.payload());
           addAlbumImage(albumInfo);
           if (++index === maxAlbums) return; // increment index, and stop once it reaches 5
           downloadNextAlbumImage(); // start downloading the next album image
       );
       imageReq.send();
  }
  function addAlbumImage(albumInfo){ // adds new item node to list, and album image node to item
       let listItem = document.createElement("li");
       HTMLList.appendChild(listItem);
       if (!albumInfo["found"]) return; // if image not found, skip adding image node
       let img = document.createElement("img");
       img.setAttribute("src", albumInfo["url"]);
       listItem.appendChild(img);
  }
}
```

Concluding Messages

🎉 The AX is back at Stanford!

Search

Stanford Report

Stanford takes back the Axe in thrilling Big Game victory

Watch next:
Stanford nears 50-year NCAA title streak

November 23rd, 2025 | 3 min read

Athletics

Stanford takes back the Axe in thrilling Big Game victory

The Cardinal dominated Cal 31–10 before a record-breaking crowd, securing its first win in the rivalry since 2020.

✍️ Gentle Reminder: Official Course Evals!

If you haven't already, we really appreciate and value your honest, anonymous feedback for AX there when you have time! ❤️ (by Dec 15th)

Your evaluations help the course, Jerry, Ben, and section leaders immensely.

- ➔ There's a **Course** tab to give feedback for Jerry / anyone or aspect of the course!
- ➔ There's a **TA / Section** tab to give feedback for me (Ben) specifically!
- ➔ In section Week 10, there was a **form at the end to give feedback to your SLs**!

Gentle Reminder: Also — take CS106B !!!

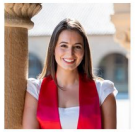
Hi there 👋 and welcome to CS106B!

CS106B Programming Abstractions is the second course in our introductory programming sequence. The prerequisite CS106A establishes a solid foundation in programming methodology and problem-solving in Python. With that under your belt, CS106B will acquaint you with the C++ programming language and introduce advanced programming techniques such as recursion, algorithm analysis, data abstraction, explore classic data structures and algorithms, and give you practice applying these tools to solving complex problems.

We're excited to share this great material with you and have a superb team of section leaders that will support you through the challenges to come. We hope you will find the time worth your investment and that you enjoy your growing mastery of the art of programming!

Teaching Team

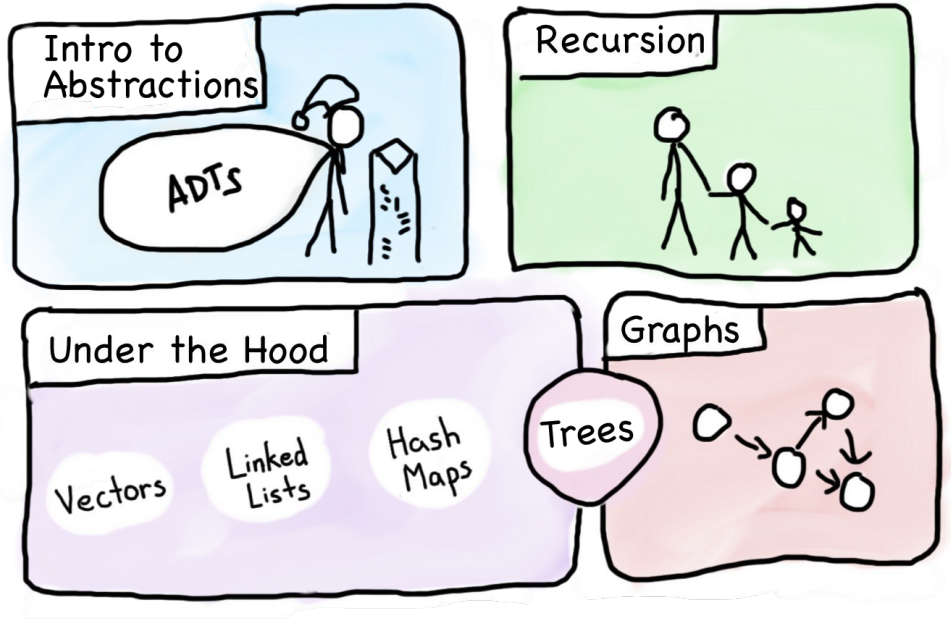
Chris Gregg



Yasmine Alonso

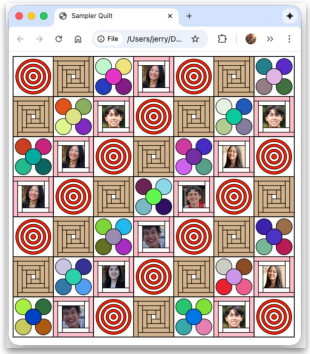
We also have many, many wonderful undergraduate Section Leaders, who teach the weekly sections for the course, grade homework and exams, and help in the LaIR.

Course description from this autumn



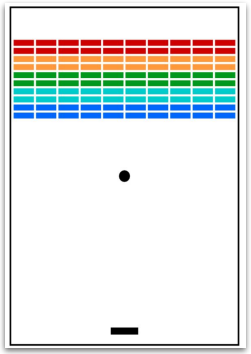
Many of our AX section leaders may be section-leading CS106B this winter!

Congrats on all the beautiful work you've done!

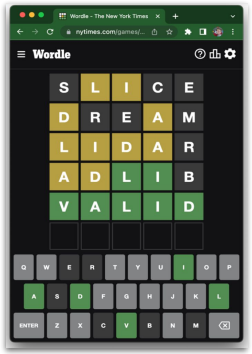


A1: JavaScript Programs

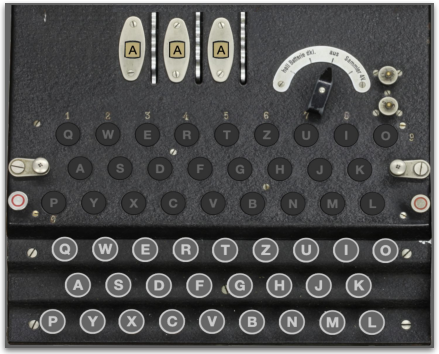
e.g., SamplerQuilt 🎨



A2: Breakout!



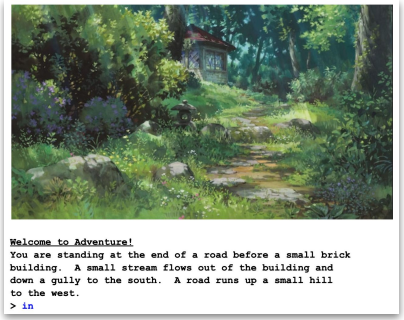
A3: Wordle



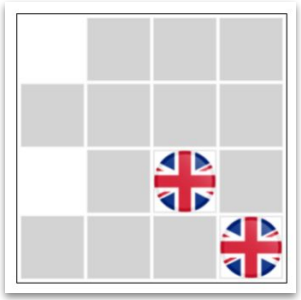
A4: Enigma Machine

A5: Python Programs

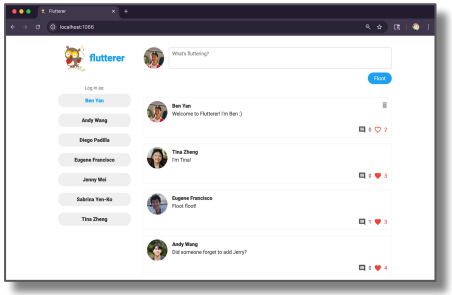
e.g., Random Sentence, Reassemble 🧬



A6: Adventure!



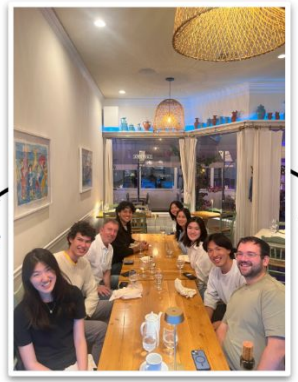
A7: Match the Flag



A8: Flutterer

We hope that you had a terrific time in AX!






Diego Emilio Padilla
106AX FOR LIFE 106AX FOR LIFE 106AX FOR LIFE
106AX FOR LIFE 106AX FOR LIFE 106AX FOR LIFE
106AX FOR LIFE 106AX FOR LIFE 106AX FOR LIFE

Jerry Cain
how did it go?

Ben Yan
I'm kinda emotional rn (I mean, not new lol),
❤️ 1

We as a staff are incredibly proud and inspired 🥹




Good luck on the final!
You're 👏 going 👏 to 👏 be 👏 awesome

- ➔ **Happy to answer any questions!** I'll walk over to Jerry's office area in CoDa E116 for office hours until ~3:30 PM if you'd like a group study space 📚❤️.
- ➔ **Exam begins 8:30 AM tomorrow morning** at CoDa B90! (all comes full circle) Bring pencils or pens, erasers, water, and any quantity of notes 🎶.

➔ **Please feel free to reach out anytime, and lmk if you need anything!**
📧**bbyan@stanford.edu**, ☎️ **507-244-0751**, or **IG at @benji.yan**