



THE UNIVERSITY OF
CHICAGO

**MASTERS IN
COMPUTATIONAL
SOCIAL SCIENCE**
THE UNIVERSITY OF CHICAGO

MACS 30111

Stacks and queues

Misc

- ↵ SE 3
- ↵ Grading ~1 week
- ↵ Final exam: proposal assignment

Topics:

- ❑ Data types and data structures review
- ❑ Interfaces and APIs
- ❑ Stacks
- ❑ Queues
- ❑ Working with Files

Data types and data structures review

- ▶ Data types: integer, float, string, Boolean
- ▶ Data structures: list, dictionary
- ▶ Custom data structures

Interface with a Python data structure

We interact with a Python data structure through its *interface* without worry about internal implementation details.

The dictionary **interface**:

```
In [1]: d = {}
```

```
In [2]: d["A"] = 4.0
```

```
In [3]: "A" in d
```

```
Out[3]: True
```

Internal implementation:

- Hash table
- Multiple steps:
 - What if a key already exists? What if it doesn't?
 - What if the hash table doesn't have enough memory allocated to add more keys?
- Abstract:
 - Interact with the interface
 - Don't need to think how to manipulate the internal hash table

Interface with a Python module

An API (***Application Programming Interface***) is a collection of functions, protocols, and tools that defines how to interact with a data structure, software library, or system, while abstracting away of the internal details.

E.g., we use the random module API to interact with *random*.

```
In [1]: import random
```

```
In [2]: random.randint(1, 100)
```

```
Out[2]: 27
```

X (twitter) API

Topics:

- ❑ Data types and data structures review
- ❑ Interfaces and APIs
- ❑ **Stacks**
- ❑ Queues
- ❑ Working with Files

Stack data structure

A **stack** is a collection of elements with a limited set of operations.

A stack supports the following **operations**:

- ↴ **Create** an empty stack `stack = []`
- ↴ **Push** a value onto the stack `stack.append(value)`
- ↴ **Pop** a value from the stack `stack.pop()`
- ↴ **Peek** at the top of the stack `stack[-1]`
- ↴ **Check** whether the stack is **empty** `len(stack) == 0`

Example stack

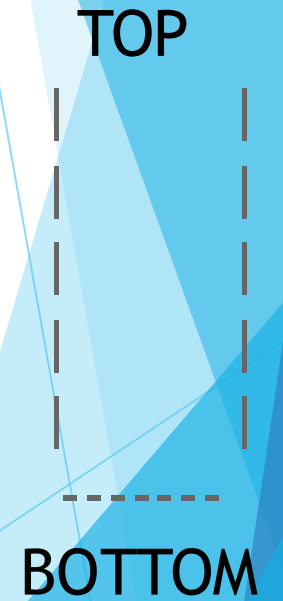
TOP

	10	
	56	
	105	
	42	
	5	

BOTTOM

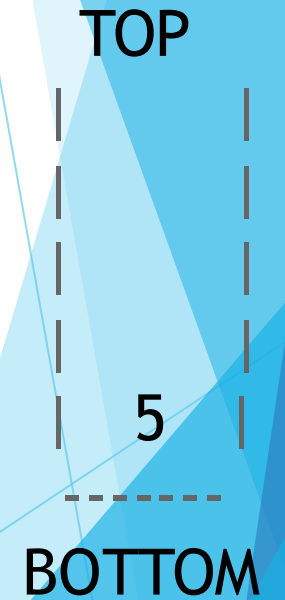
Example

1. Create an empty stack



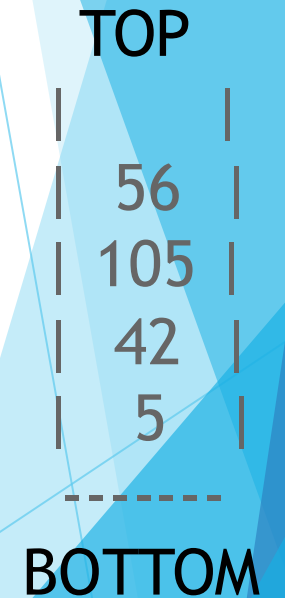
Example

1. Create an empty stack
2. Push the value 5 to the stack



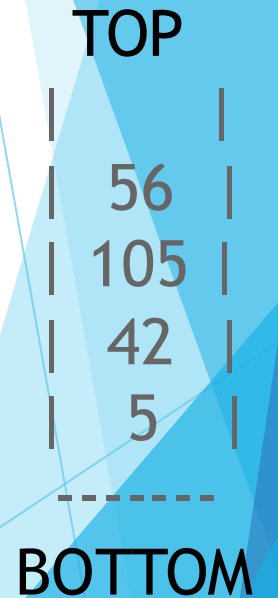
Example

1. Create an empty stack
2. Push the value 5 to the stack
3. Push the values 42, 105, and 56 to the stack



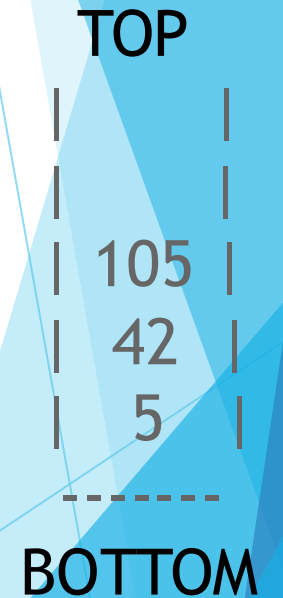
Example

1. Create an empty stack
2. Push the value 5 to the stack
3. Push the values 42, 105, and 56 to the stack
4. Pop a value from the stack



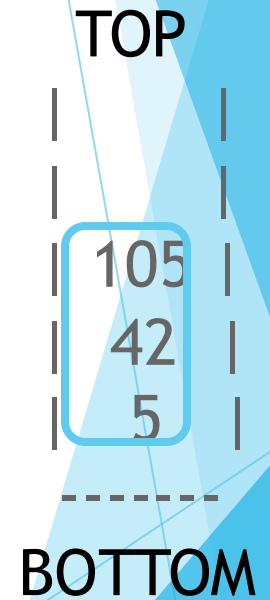
Example

1. Create an empty stack
2. Push the value 5 to the stack
3. Push the values 42, 105, and 56 to the stack
4. Pop a value from the stack
5. Peek at the stack



Example

1. Create an empty stack
2. Push the value 5 to the stack
3. Push the values 42, 105, and 56 to the stack
4. Pop a value from the stack
5. Peek at the stack
6. Check whether the stack is empty



API revisited

Recall that we interact with a data type in Python through its API.

User interface:

```
In [1]: d = {}  
  
In [2]: d["A"] = 4.0  
  
In [3]: "A" in d  
Out[3]: True
```

API

Developer implementation details:

```
"""  
Hash table,  
functions, and other  
dictionary  
implementation  
details  
"""
```

Stack implementation

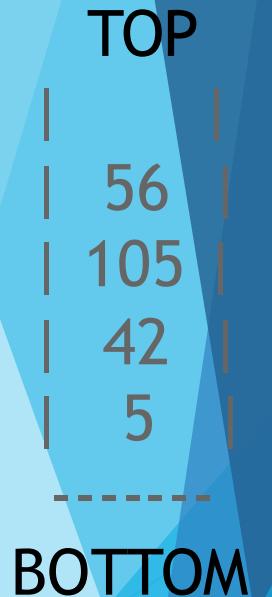
Implement a function-based interface for a stack.

```
def stack_create():  
    return []
```


Stack implementation

Implement a function-based interface for a stack.

```
def stack_create():  
    return []  
  
def stack_push(stack, value):  
    stack.append(value)  
  
def stack_pop(stack):  
    return stack.pop()  
  
def stack_top(stack):  
    return stack[-1]  
  
def stack_is_empty(stack):  
    return len(stack) == 0
```



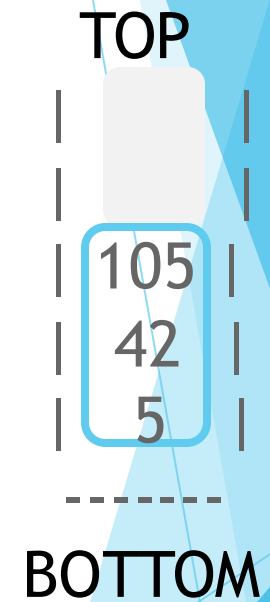
String representation

It's often a good idea to add a function to visualize a data structure.

```
def stack_to_string(stack):  
    s = " TOP OF THE STACK\n"  
    s += "-----\n"  
  
    for v in reversed(stack):  
        s += str(v).center(20) + "\n"  
  
    s += "-----\n"  
    s += "BOTTOM OF THE STACK\n"  
    return s
```

Example

1. Create an empty stack
2. Push the value 5 to the stack
3. Push the values 42, 105, and 56 to the stack
4. Pop a value from the stack
5. Peek at the stack
6. Check whether the stack is empty



First In Last Out

Coding practice: 2.3.2

YOUR TURN!!

1. **Create** an empty stack named s111
2. **Push** the value 42 to the stack
3. **Push** the values 5, 7, and 12 to the stack
4. **Pop** a value from the stack
5. **Pop** a value from the stack
6. **Peek** at the stack
7. **Check** whether the stack is **empty**

```
def stack_create():  
    return []  
  
def stack_push(stack, value):  
    stack.append(value)  
  
def stack_pop(stack):  
    return stack.pop()  
  
def stack_top(stack):  
    return stack[-1]  
  
def stack_is_empty(stack):  
    return len(stack) == 0
```

Python modules

Once we define an API for a data structure, we can put it in a **Python module** and **import** it from IPython or other Python files.

mystack.py

```
def stack_create():  
    return []  
  
def stack_push(stack, value):  
    stack.append(value)  
  
# stack operations
```

myprogram.py

```
import mystack  
  
s = mystack.stack_create()  
  
mystack.stack_push(s, 5)
```

IPython

```
In [1]: import mystack  
  
In [2]: s = mystack.stack_create()  
  
In [3]: mystack.stack_push(s, 5)
```

Import module from a different path:

```
import sys  
sys.path.append("/path/to/my/modules/")  
import my_module
```

Topics:

- ❑ Data types and data structures review
- ❑ Interfaces and APIs
- ❑ Stacks
- ❑ **Queues**
- ❑ Working with files

Queue data structure

A **queue** is a collection of elements with a limited set of operations.

Queue **operations**:

- ↴ **Create** an empty queue `queue = []`
- ↴ **Enqueue** a value at the back of the queue `queue.append(value)`
- ↴ **Dequeue** a value from the front of the queue `queue.pop(0)`
- ↴ **Peek** at the front of the queue `queue[0]`
- ↴ **Check the size** of the queue `len(queue) == 0`

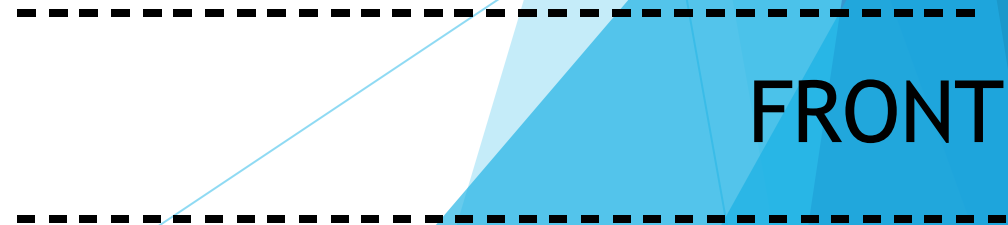


Example

1. Create an empty queue

BACK

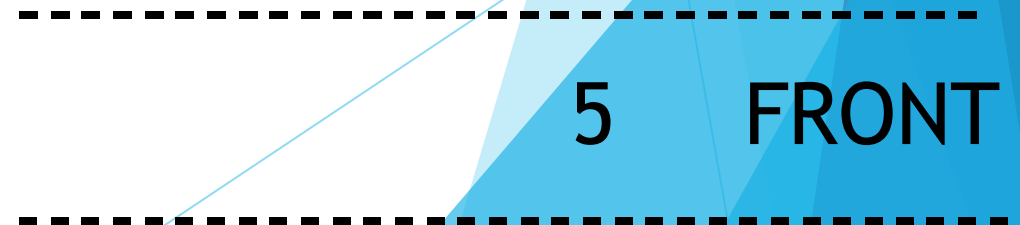
FRONT



Example

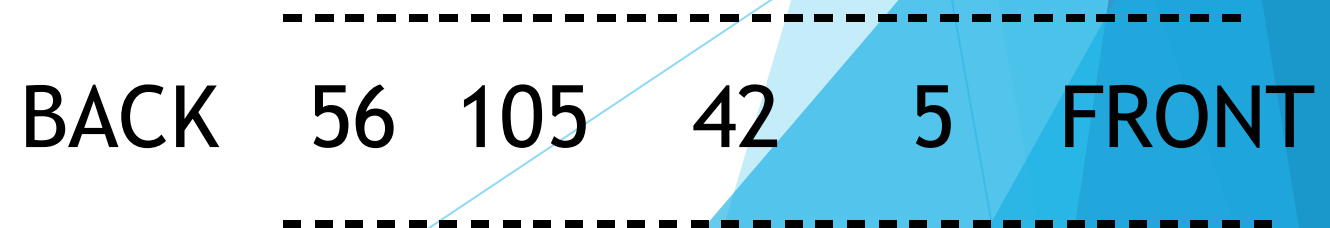
1. Create an empty queue
2. Enqueue the value 5 to the queue

BACK



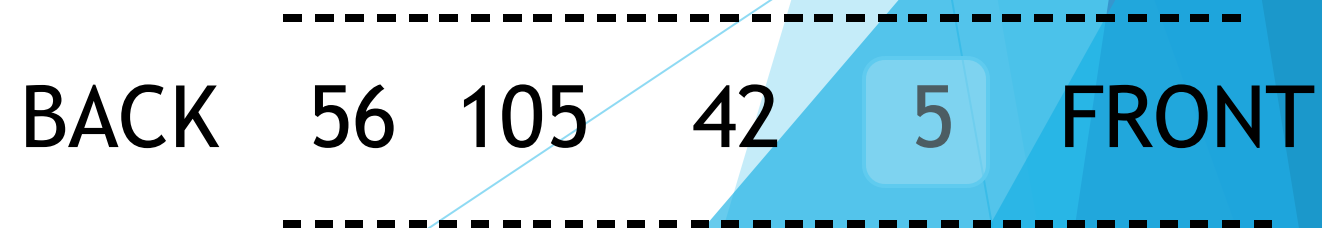
Example

1. Create an empty queue
2. Enqueue the value 5 to the queue
3. Enqueue the values 42, 105, and 56 to the queue



Example

1. Create an empty queue
2. Enqueue the value 5 to the queue
3. Enqueue the values 42, 105, and 56 to the queue
4. Dequeue a value from the queue



Example

1. Create an empty queue
2. Enqueue the value 5 to the queue
3. Enqueue the values 42, 105, and 56 to the queue
4. Dequeue a value from the queue
5. Peek at the front of the queue

BACK

56

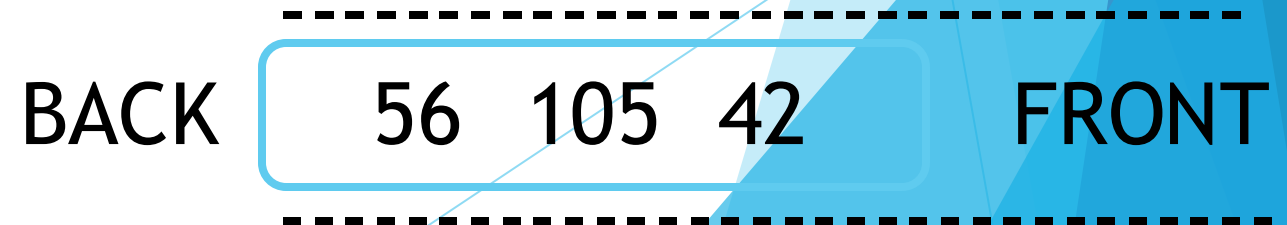
105

42

FRONT

Example

1. Create an empty queue
2. Enqueue the value 5 to the queue
3. Enqueue the values 42, 105, and 56 to the queue
4. Dequeue a value from the queue
5. Peek at the front of the queue
6. Check the size of the queue



Queue implementation

We will implement a queue as a list in Python.

Should the front of the queue be the end of the list?

```
q = [56, 105, 42, 5]
```

Should it be the beginning of the list?

```
q = [5, 42, 105, 56]
```

BACK 56 105 42 5 FRONT

Complexity and efficiency of list operations

Inserting or deleting **from the beginning** of a list is an **expensive** operation since all elements must be shifted.

Insert the value 3 to the beginning of the list:

```
lst = [2, 4, 8]
```

```
lst = [3, 2, 4, 8]
```

Delete the value from the beginning of the list:

```
lst = [2, 4, 8]
```

```
lst = [4, 8]
```

Time Complexity

Complexity and efficiency of list operations

Appending or deleting **from the end** of a list is a **simple** operation.

Append the value 3 to the end of the list:

```
lst = [2, 4, 8]
```

```
lst = [2, 4, 8, 3]
```

Delete a value from the end of the list:

```
lst = [2, 4, 8]
```

```
lst = [2, 4]
```

Time complexity

Queue implementation

What if we make the *beginning* of the list the *back* of the queue?

Enqueue the value 2:

[8, 9, 17]

back

front

[2, 8, 9, 17]

Dequeue a value:

[2, 8, 9, 17]

[2, 8, 9]

What if we make the *beginning* of the list the *front* of the queue?

Enqueue the value 2:

front [17, 9, 8] back

[17, 9, 8, 2]

Dequeue a value:

[17, 9, 8, 2]

[9, 8, 2]

Queue implementation



Now we can implement a function-based interface for a queue.

```
def queue_create():  
    return []  
  
def queue_is_empty(queue):  
    return len(queue) == 0  
  
def queue_length(queue):  
    return len(queue)  
  
def queue_enqueue(queue, value):  
    queue.append(value)  
  
def queue_dequeue(queue):  
    return queue.pop(0)  
  
def queue_front(queue):  
    return queue[0]
```

```
def queue_to_string(queue):  
    s = "FRONT OF THE QUEUE\n"  
    s += "-----\n"  
  
    for v in queue:  
        s += str(v).center(19) + "\n"  
  
    s += "-----\n"  
    s += "BACK OF THE QUEUE \n"  
    return s
```

YOUR TURN!!

1. **Create** an empty queue named q111
2. **Enqueue** the value 42 to the stack
3. **Enqueue** the values 5, 7, and 13 to the queue
4. **Dequeue** a value from the queue
5. **Peek** at the front of the queue
6. **Check** the **size** of the queue
7. **Check** whether the queue is **empty**

```
def queue_create():  
    return []  
  
def queue_is_empty(queue):  
    return len(queue) == 0  
  
def queue_length(queue):  
    return len(queue)  
  
def queue_enqueue(queue, value):  
    queue.append(value)  
  
def queue_dequeue(queue):  
    return queue.pop(0)  
  
def queue_front(queue):  
    return queue[0]
```

First In First Out vs Last In First Out

Coding practice: 2.3.3

Python modules

Once we define an API for a data structure, we can put it in a **Python module** and **import** it from IPython or other Python files.

myqueue.py

```
def queue_create():  
    return []  
  
def queue_enqueue(queue, value):  
    queue.append(value)  
  
def queue_dequeue(queue):  
    return queue.pop(0)  
  
# queue operations
```

myprogram.py

```
import myqueue  
  
q = myqueue.queue_create()  
mystack.queue_enqueue(q, 5)
```

IPython

```
In [1]: import myqueue  
  
In [2]: q = myqueue.queue_create()  
  
In [3]: myqueue.enqueue(q, 5)
```

Import module from a different path:

```
import sys  
sys.path.append("path-to/location/")  
import myqueue
```

Examples of use

↓ **Stacks:** last in, first out

- ↪ `undo` functions (e.g. text writing, back button)
- ↪ Check for matching (e.g. matching parentheses)
- ↪ Function calls

↓ **Queues:** first in, first out

- ↪ Scheduling
- ↪ Breadth-first scheduling in networking
- ↪ Event handling with GUI



Final project: prep

Application: demonstrate mastery of python

- ▶ Holistic: showcase learning over course

Working with files

Common Programming Pattern

Common pattern when working with data:

1. **Read** the contents of a file (or files) from disk and **load** the data into one or more data structures
2. **Manipulate** the data in some way
3. **Print** the result or **write** the data back to disk

Sample application

Given a file of email addresses (username@domain), construct a file with the corresponding user names.

instructor-email.txt

```
amr@cs.uchicago.edu  
borja@cs.uchicago.edu  
yanjingl@cs.uchicago.edu  
mwachs@cs.uchicago.edu  
dupont@cs.uchicago.edu
```



instructor-email-sorted.txt

```
["amr@cs.uchicago.edu",  
 "borja@cs.uchicago.edu",  
 "dupont@cs.uchicago.edu",  
 "mwachs@cs.uchicago.edu",  
 "yanjingl@cs.uchicago.edu"]
```

Coding practice: 4.1.1

Common Programming Pattern

Common pattern when working with data:

1. **Read the contents of a file (or files) from disk and load the data into one or more data structures**
2. **Manipulate** the data in some way
3. **Print** the result or **write** the data back to disk

Opening a file



Basic File I/O

To **access** the contents of a file, we first need to open it:

```
f = open("instructor-email.txt")
```

file pointer

To **read data** from a file, we use the read method:

```
addrs = f.read()
```

read the entire contents into a string

When we are done with a file, we need to close it:

```
f.close()
```

close the file pointer

Coding practice: 4.1.1

Alternative to *close()*

The ***with*** statement to ensure that a file is closed once we're done with it:

```
with open("instructor-email.txt") as f:  
    s = f.read()  
    email_addresses = sorted(s.split())
```

Read the file one line at a time

Use a *for* loop to iterate over a text file line by line:

```
with open("instructor-email.txt") as f:
    for line in f:
        print(line)
```

```
with open("instructor-email.txt") as f:
    for line in f.readlines():
        print(line)
```

extra empty line

line.strip()

Coding practice: 4.1.1

Common Programming Pattern

Common pattern when working with data:

1. **Read** the contents of a file (or files) from disk and **load** the data into one or more data structures
2. **Manipulate** the data in some way
3. **Print the result or write the data back to disk**

Write data to a file

To write to a file, we must open the file in **write mode**.

```
with open("names.txt", "w") as f:
    f.write("Anne Rogers\n")
    f.write("Borja Sotomayor\n")
    f.write("Yanjing Li\n")
    f.write("Matthew Wachs\n")
    f.write("Todd Dupont\n")
```

We can also use ***print*** to avoid having to worry about the newline.

```
with open("names2.txt", "w") as f:
    print("Anne Rogers", file=f)
    print("Borja Sotomayor", file=f)
    print("Yanjing Li", file=f)
    print("Matthew Wachs", file=f)
    print("Todd Dupont", file=f)
```

Very important:

- Opening an existing file in write mode will **wipe all its contents!**
- Opening a file that does not exist in write mode will **create the file.**

Coding practice: 4.1.2

Summary

The common programming pattern:

1. Load the data from disk:
 - a. **Open** a file to **read**
 - b. Read the contents of the file from disk
 - c. Load the data into a data structure
2. Manipulate the data in some way
3. Print the result or write the data back to disk
 - a. **Write** the data
 - b. **Close** the file (or use a with statement when you open it)

Topics:

- ❑ Basic file I/O
 - ❑ Open: load the data from disk
 - ❑ Read: manipulate the data
 - ❑ Close: print the results or write the data back to disk
- ❑ **Working with tabular data using CSV files**
- ❑ Working with JSON files
- ❑ Other file formats

CSV (Comma Separated Values) format

CSV files are useful for storing **tabular data**: any data that can be organized into rows, each with the same columns (or "fields")

`instructors.csv`

`id,lname,fname,email`

header

`hdambane,Dambanemuya,Henry,hdambane@uchicago.edu`

`dapeterson,Peterson,David,dapeterson@uchicago.edu`

`zwang13,Wang,Zhao,zwang13@uchicago.edu`

`jclindaniel,Clindaniel,Jon,jclindaniel@uchicago.edu`

`nardin,Nardin,Sabrina,nardin@uchicago.edu`

Exercise

- ⌵ You are working on a project creating a directory of buildings based on where MACSS classes typically meet.
- ⌵ Use text file: <https://uchicago.box.com/s/kp207rd2vita1a4zz6749oe8jkkvk85l6>
- ⌵ How would you load your data?
- ⌵ Next, you want to select the following buildings: `sched = ["1155", "SS", "TTI", "K"]` and output the new list into a separate file
- ⌵ How would you summarize or use the data?

What if you want to skip lines?

- ▶ Multiple ways to approach:

```
new_lst = []
```

```
with open('names.txt') as names:
```

```
    next(names)
```

```
    next(names)
```

```
    next(names)
```

```
new_lst = names.readlines()
```

```
new_list = []
```

```
with open('names.txt') as names:
```

```
    for i, line in enumerate(names):
```

```
        if i > 3 and i < 149:
```

```
            new_list.append(line)
```

Read file using csv module

- ▶ *csv.DictReader* - read rows from a CSV file into dictionaries
- ▶ *csv.DictWriter* - write dictionaries into rows of a CSV file

Alternatively, we could also use:

- ▶ *csv.reader* - read rows from a CSV file into a list of lists
- ▶ *csv.writer* - write lists into rows of a CSV file

Different ‘modes’

- ▶ You can open files in different ‘modes’
 - ▶ r: ‘read’ mode (default)
 - ▶ w: ‘write’ mode (needs specified)
 - ▶ a: append

If you’re just reading a file, you can operate as normal. If you’re wanting to write a new file, *then* you will use “w”.

DANGER ALERT!!! In “w” mode, you will OVERWRITE THE PREEXISTING FILE!!

Writing files

“w” for “write mode”

```
with open("names_cleaned.txt", "w") as f:  
    for build in new_list:  
        print(build, file=f)
```

Bringing the exercise together:

making it pretty:

```
def get_buildings(input_filename, output_filename, sched):
```

```
'''
```

extract relevant buildings from campus list

Inputs:

input_filename: (string) name of a file with buildings

output_filename: (string) name for the output file.

```
'''
```

Load data into a data structure (a list of strings)

```
sched = ["1155", "SS", "TTI", "K"]
```

```
buildings = []
```

```
with open("input_filename.txt") as f:
```

```
    for line in f:
```

```
        builds = line.strip().split("\t", 1)
```

```
        buildings.append(builds)
```

Transform the data

```
buildings_select = []
```

```
for line in buildings:
```

```
    if line[0] in sched:
```

```
        buildings_select.append(line)
```

Write the data

```
with open("output_filename.txt", "w") as f:
```

```
    for build in buildings_select:
```

```
        print(build, file=f)
```



The background features abstract, overlapping geometric shapes in various shades of blue, ranging from light sky blue to deep navy blue. These shapes are primarily located on the left and right sides of the frame, creating a modern, dynamic feel. The central area is a plain, light grayish-white.

Exam prep!

Exam prep: spot 3 errors and rewrite code

making it pretty:

```
def get_buildings(input_filename, output_filename, sched):
```

```
'''
```

extract relevant buildings from campus list

Inputs:

input_filename: (string) name of a file with buildings

output_filename: (string) name for the output file.

```
'''
```

Load data into a data structure (a list of strings)

```
sched = ["1155", "SS", "TTI", "K"]
```

```
buildings = []
```

```
with open(input_filename) as f:
```

```
    for line in f:
```

```
        builds = line.strip().split("\t", 1)
```

```
        buildings.append(builds)
```

Transform the data

```
buildings_select = []
```

```
for line[0] in buildings:
```

```
    if line in sched:
```

```
        buildings_select.append(line)
```

Write the data

```
with open(output_filename) as f:
```

```
    for build in buildings:
```

```
        print(build, file=f)
```

Recap:

- ❑ Data structures:
 - ❑ **Stacks:** when you need last in, first out. Specific methods to add to/from the TOP
 - ❑ **Queues:** when you need first in, first out. Specific methods to add to the BACK of the queue
- ❑ Working with files!