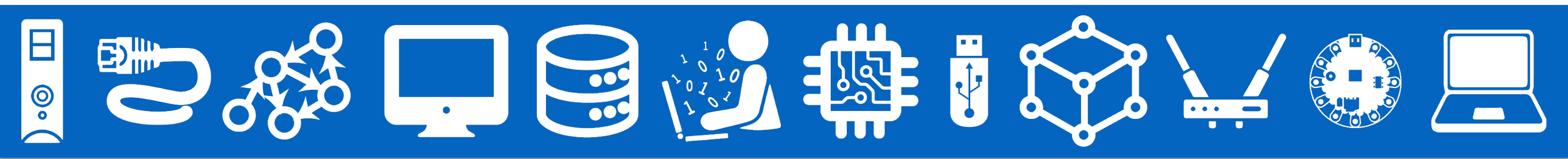
# CS134:
# Range & Nested Loops



# Announcements & Logistics

- **Lab 3** sessions are today/tomorrow; due Wednesday/Thursday
  - More involved than previous labs, so please use help hours
  - Reminder: do **NOT** use utilities not discussed in class
    - We've carefully designed the labs to require only functions & concepts *discussed in class meetings*
    - We've intentionally ordered material to emphasize algorithmic thinking and benefit your development as a computer scientist rather than as a Python-specific programmer
      - This means no methods using **dot.notation()**! (Why?)
- **HW 3** due tonight at 10pm on Gradescope

**Do You Have Any Questions?**

# Last Time

- `for`..Loops allow us to look at each element in a sequence
    - The **loop variable** defines what the name of that element will be in the loop
    - An optional **accumulator variable** is useful for keeping a running tally of properties of interest
        - Indentation works the same as with if--statements: if it's indented under the loop, it's executed as part of the loop
- Can extract subsequences using `[start:end:step]` syntax (slicing)

Different problems may require different decisions with respect to loop variables, accumulator variables, and whether you need to index/slice or not!

# Today's Plan

- New sequence type: `range`
- Explore different combinations of loops
  - Loop(s) within a loop (called nesting)

# Review: Sequences in Python

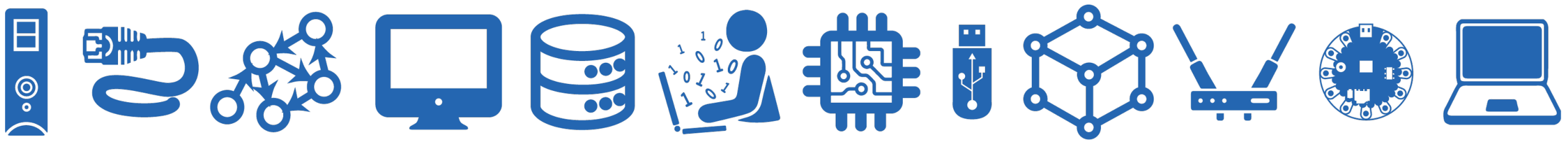
- **Sequences** in Python represent **ordered collections of elements**: e.g., lists, strings, ranges, etc.
- Strings are immutable sequences of characters
- Ranges are immutable sequences of numbers
- Lists can be **heterogenous** (strings, ints, floats, etc)
    - Example: `my_list = ["Hello", 42, 23.5, True]`
    - In CS, we use **zero-indexing**, so we say that `'Hello'` is at **index** 0, `42` is at **index** 1, and so on
- We can access each character of a list using these **indices**

# Summary: Sequence Operations

| Operation | Result |
|---|---|
| `seq[i]` | The `i`'th item of `seq`, when starting with 0 |
| `seq[si:ee]` | slice of `seq` from `si` to `ee` |
| `seq[si:ee:s]` | slice of `seq` from `si` to `ee` with step `s` |
| `len(seq)` | length of `seq` |
| `seq1 + seq2` | The concatenation of `seq1` and `seq2` |
| `x in seq` | True if `x` is contained within `seq` |
| `x not in seq` | False if `x` is contained within `seq` |

All of these operators work on both **strings** and **lists**!

# Ranges



# Ranges (another sequence!)

- Python provides an easy way to iterate over numerical sequences using the **range** data type, which is **another sequence**
- When the `range()` function is given two integer arguments, it returns a ***range object*** of all integers starting at the first and up to, *but not including*, the second  (note: default starting value is 0)
- To see the values included in the range, we can pass our range to the `list()` function which returns a **list** of them

```
>>> range(0, 10)
range(0, 10)
```

```
>>> list(range(0, 10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> type(range(0, 10))
range
```

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

# Ranges (another sequence!)

- Python provides an easy way to iterate over numerical sequences using the **range** data type, which is **another sequence**
- When the `range()` function is given two integer arguments, it returns a ***range object*** of all integers starting at the first and up to, *but not including*, the second (note: default starting value is 0)
- To see the values included in the range, we can pass it to the `list()` function which returns a **list** of them

To see elements in range, pass range to `list()` function

```
>>> range(0, 10)
range(0, 10)
```

```
>>> list(range(0, 10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> type(range(0, 10))
range
```

A range is a **type** of sequence in Python (like string and list)

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

First argument omitted, **defaults to 0**

# Iterating Over Ranges

```
# what does this print?

for i in range(5):
    print('$' * i)
```

```
                i = 0
 $              i = 1
 $$             i = 2
 $$$            i = 3
 $$$$           i = 4
```

- In addition to iterating over strings and lists, we can use a **for loop** and a range to simply **repeat** a task.
- This loop print a pattern to the screen.

Looks a lot like `[0, 1, 2, 3, 4]`

# Using Range For Parallel Iteration

- This also a really convenient way for iterating over two lists in parallel
- Say we wanted to iterate over two lists
- `chars = ['a', 'b', 'c']` and `nums = [1, 2, 3]`
- And form a new list `['a1', 'b2', 'c3']`
- Here's how we'd do it

```
chars = ['a', 'b', 'c']
nums = [1, 2, 3]

# initialize accumulation variable
# for each item in chars
    # add current char to matching num
    # accumulate in a list
```

```
>>> char_nums
['a1', 'b2', 'c3']
```

# Using Range For Parallel Iteration

- This also a really convenient way for iterating over two lists in parallel
- Say we wanted to iterate over two lists
- `chars = ['a', 'b', 'c']` and `nums = [1, 2, 3]`
- And form a new list `['a1', 'b2', 'c3']`
- Here's how we'd do it

```
chars = ['a', 'b', 'c']
nums = [1, 2, 3]
char_nums = []

for i in range(0, len(chars)):
    cnum = chars[i] + str(nums[i])
    char_nums = char_nums + [cnum]
```

Loop Variable

Accumulator Variable

```
>>> char_nums
['a1', 'b2', 'c3']
```

# Using **`range`** to check **`palindromes`**

- Write a function that iterates over a given list of strings `word_list`, returns a (new) list containing all the strings in `word_list` that start and end with the same character (ignoring case).
- Last class we saw the `word == word[::-1]` solution
- *Another solution:* Compare the first and last character, the second and second-to-last character, etc.

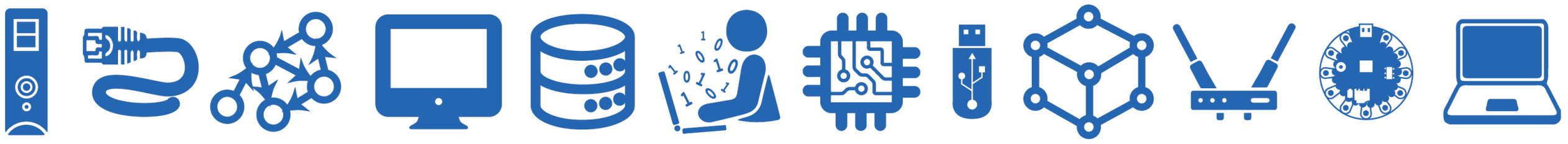
```
def is_palindrome_range(word) :
    # Test by comparing pairs of characters one at a time.
    # Since we need to compare each char in the "first half"
    # to corresponding char in the "second half",
    # we need to execute len(string) // 2 comparisons
    for i in range(len(word) // 2) :
        if word[i] != word[-(i+1)] :
            return False
    return True
```

# Using **`range`** to check **`palindromes`**

- Last class we saw the **`word == word[::-1]`** solution
- *Another solution:* Compare the first and last character, the second and second-to-last character, etc.
- Why do this solution?

```
def is_palindrome_range(word) :
    # Test by comparing pairs of characters one at a time.
    # Since we need to compare each char in the "first half"
    # to corresponding char in the "second half",
    # we need to execute len(string) // 2 comparisons
    for i in range(len(word) // 2) :
        if word[i] != word[-(i+1)] :
            return False
    return True
```

# Nested Loops



# Nested Loops

- A **for loop** body can contain one (or more!) additional **for loops**:
    - Called **nested for loops**
    - Conceptually similar to nested conditionals
- Example: What do you think is printed by the following Python code?

```
# What does this do?
def mystery_print(word1, word2):
    '''Prints something'''
    for char1 in word1:
        for char2 in word2:
            print(char1 + char2)
```

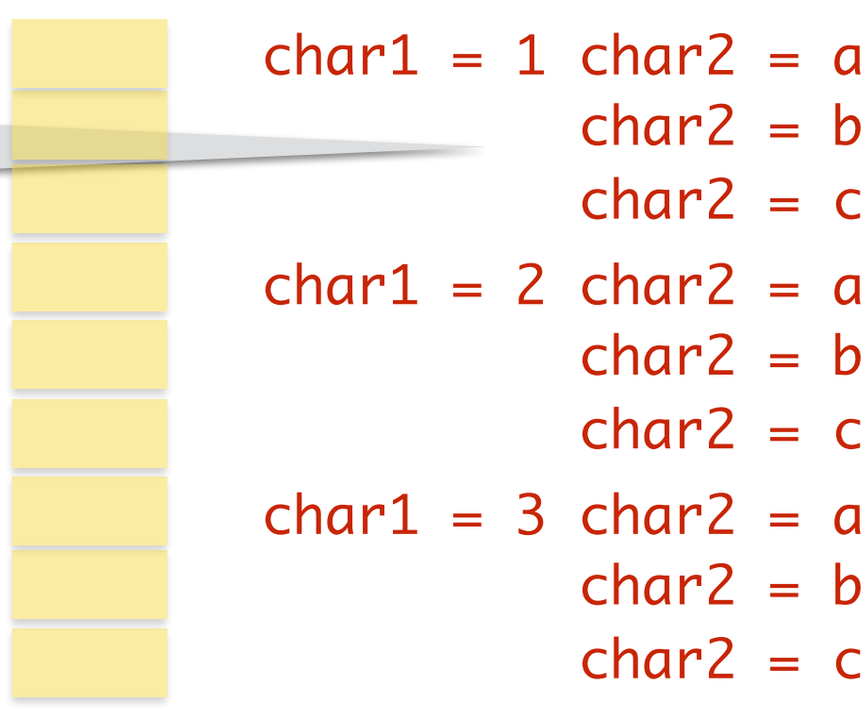
```
mystery_print('123', 'abc')
```

```python
# What does this do?
def mystery_print(word1, word2):
    '''Prints something'''
    for char1 in word1:
        for char2 in word2:
            print(char1 + char2)
```

```python
mystery_print('123', 'abc')
```

```
1a
1b
1c
2a
2b
2c
3a
3b
3c
```

Inner loop (w/ char2 and word2) runs to completion on **each iteration** of the outer loop



# Nested Loops

- What is printed by the nested loop below?

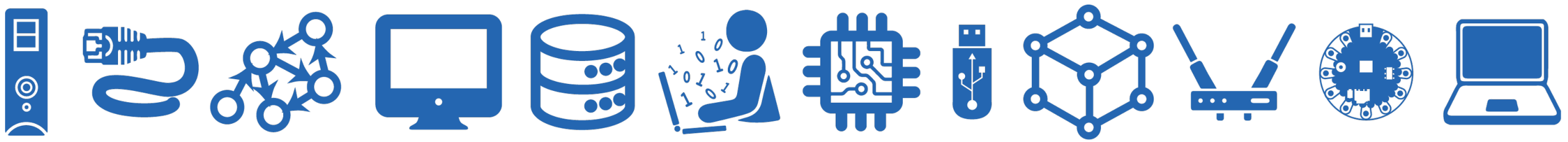
```
# What does this print?
for letter in ['b', 'd', 'r', 's']:
    for suffix in ['ad', 'ib', 'ump']:
        print(letter + suffix)
```

```
# What does this print?
for letter in ['b', 'd', 'r', 's']:
    for suffix in ['ad', 'ib', 'ump']:
        print(letter + suffix)
```

```
letter = 'b'      suffix ='ad'      bad
                          'ib'      bib
                         'ump'     bump

letter = 'd'      suffix ='ad'      dad
                          'ib'      dib
                         'ump'     dump

letter = 'r'      suffix ='ad'      rad
                          'ib'      rib
                         'ump'     rump

letter = 's'      suffix ='ad'      sad
                          'ib'      sib
                         'ump'     sump
```

Inner loop (w/ suffixes) runs to completion on **each iteration** of the outer loop (w/ prefixes)

# Nested Loops and Ranges



# Loops and Ranges to Print Patterns

We previously used a single **for loop** and a single range to **repeat** a task.

- What if we had multiple for loops and multiple ranges? The following loops print a pattern to the screen. (Look closely at the indentation!)

```
# what does this print?

for i in range(5):
    print('$' * i)
for j in range(5):
    print('*' * j)
```

```
# what does this print?

for i in range(5):
    print('$' * i)
    for j in range(i):
        print('*' * j)
```

**What are the values of i and j???**

# Iterating Over Ranges

```
# what does this print?

for i in range(5):
    print('$' * i)
for j in range(5):
    print('*' * j)
```

**We've seen this for loop and pattern before**

**Same pattern, but with '*' instead**

```
          i = 0
$         i = 1
$$        i = 2
$$$       i = 3
$$$$      i = 4

          j = 0
 *        j = 1
 **       j = 2
 ***      j = 3
 ****     j = 4
```

These for loops are **sequential**.
One follows **after** the other.

# Iterating Over Ranges

```
# what does this print?

for i in range(5):
    print('$' * i)
for j in range(5):
    print('*' * j)
```

```
          i = 0
$         i = 1
$$        i = 2
$$$       i = 3
$$$$      i = 4

          j = 0
*         j = 1
**        j = 2
***       j = 3
****      j = 4
```

```
# what does this print?

for i in range(5):
    print('$' * i)
    for j in range(i):
        print('*' * j)
```

```
          i = 0
$         i = 1
              j = 0
$$        i = 2
              j = 0
*             j = 1
$$$       i = 3
              j = 0
*             j = 1
**            j = 2
$$$$      i = 4
              j = 0
*             j = 1
**            j = 2
***           j = 3
```

On right, for loops are **nested**.
One loop is **inside** the other.

# Iterating Over Ranges

```
# what does this print?

for i in range(5):
    print('$' * i)
    for j in range(i):
        print('*' * i)
```

i, not j!

```
$
*
$$
**
**
$$$
***
***
***
$$$$
****
****
****
****
```

```
i = 0
i = 1
j = 0
i = 2
j = 0
j = 1
i = 3
j = 0
j = 1
j = 2
i = 4
j = 0
j = 1
j = 2
j = 3
```

```
# what does this print?

for i in range(5):
    print('$' * i)
    for j in range(i):
        print('*' * j)
```

```
$
$$
$$$
$$$$
```

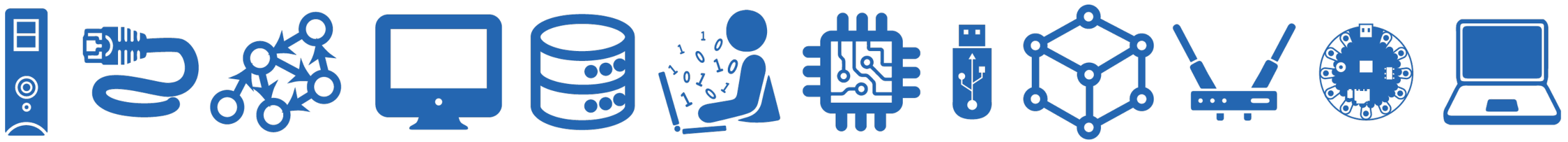
```
i = 0
i = 1
j = 0
i = 2
j = 0
j = 1
i = 3
j = 0
j = 1
j = 2
i = 4
j = 0
j = 1
j = 2
j = 3
```

# Range: Take-Aways

- `range` objects are a new type of sequence that are essentially a *range* of integers
    - `range(0, 10)` will generate a sequence of integers from 0 to 10 (exclusive of 10)
- Using `range` objects with `for` loops allows us to:
    - produce behaviors based on counts of times through a loop
    - iterate over sequences using their indices
    - iterate over the objects in two sequences simultaneously

Different problems will require different decisions with respect to sequences and whether you need a `range` or to iterate directly over a sequence!

# Flag Variables



# Starting/Stopping Accumulating

- We've seen many different examples using **accumulator variables** to track values of interest:
    - Counting the number of vowels
    - Maintaining a list of all palindromes, etc.
- However, so far, we've only seen accumulating over an *entire* sequence
- What if we wanted to only count/track *part* of a sequence's values?

**Flag Variables!**

"location" often implies **index**, which often means using `range()`

# Flag Variables

- Identifying the first locations of a character in each string from a list:
  - *Let's decompose the problem first!*

```
def first_locations_of(char, list_of_str):
    """ Returns a list containing the index
        where char first appears within each list_of_str
    """
    # initialize accumulator variable
    # for each word in the list of strings
        # initialize our FLAG VARIABLE - as "not found"
        # for each index in the word
            # if we haven't yet found our character, but we just did
                # add this location to our accumulator
                # update the FLAG VARIABLE to be "found"
    # return our accumulator variable
```

```
>>> first_locations_of('e', ["eat", "more", "vegetables"])
[0, 3, 1]
```

# Flag Variables

- Identifying the first locations of a character in each string from a list:

```
def first_locations_of(char, list_of_str):
    """ Returns a list containing the index
        where char first appears within each list_of_str
    """
    locations = []
    for word in list_of_str:
        found = False # starting new word, haven't found it!
        for i in range(len(word)):
            if not found and word[i] == char: # we found it
                locations = locations + [i]
                found = True
    return locations
```

```
>>> first_locations_of('e', ["eat", "more", "vegetables"])
[0, 3, 1]
```

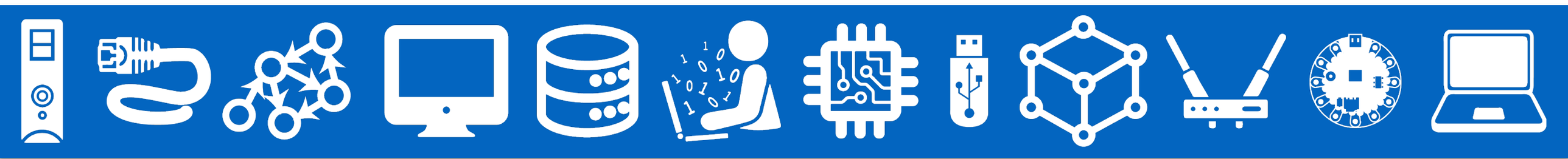
# Flag Variables

- Can often be used as a way to record the "state" that the program is in
  - `is_found` and `not is_found` is a simple on/off state...
  - But other problems could require other (or more!) states!
  - Other (or more!) flag variables!
- Like **accumulator variables**, **flag variables** are optional
- We only use them when the problem calls for them
  - Appropriate use of accumulator and flag variables requires ***computational thinking***

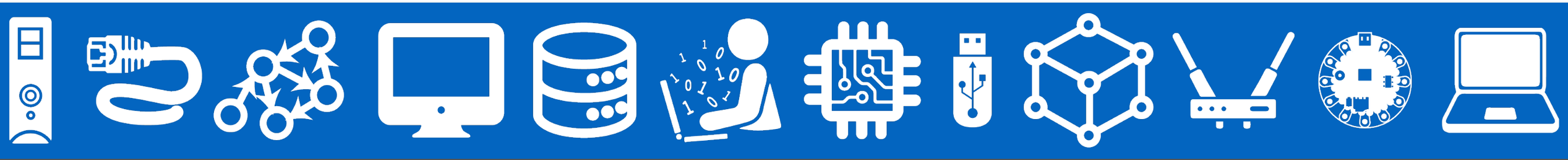
# Summary

- **Range** is a flexible sequence type often used for indexing or for executing a loop a certain number of times
- **Loops can be nested** inside other loops
  - Inner loops execute once *per iteration* of their containing loop
- **Accumulator variables** can be used to track and/or store values of interest
- **Flag variables** are used to determine when to begin or stop tracking something
  - Helps record the "state" of the program

# The end!



# CS134:
# Lab 3



# Lab 3: Goals

- In this lab, you will accomplish three tasks:
  - Construct a **module** of tools for examining `strings` (in `madlibs.py`)
  - Test your toolbox using simple test cases in `runtests.py`
  - Reuse parts of our toolbox to solve various Madlibs.
- You will gain experience with the following:
  - **Sequences** (**strings**, **lists,** and **ranges**), and associated **operators**
  - Writing simple and nested **for loops**

# Testing Functions: `runtests.py`

- We have already seen two ways to test a function
  - You can run your code 1) interactively or 2) as a script
- Last week, we started using a separate file, `runtests.py`, to test our code as a script
    - To do this, `runtests.py` has to import our functions that we implemented in our main lab file
    - Then, we define functions in `runtests.py` to call the functions we implemented in the lab, such as `is_prefix()`
    - Remember: We must *call* functions for them to be executed!
- To ensure that the tests are not run in interactive Python, we place this command within a "guarded" if block:

```
if __name__ == '__main__':
```

# Testing Functions: `runtests.py`

```
from text_utils import read_stringlist_from_file, format_madlib
from madlibs import is_prefix, is_suffix, all_text_after, \
    get_madlibs_replacement, solved_madlibs
```

**Imports your functions from `madlibs.py`**

```
def is_prefix_test1() :
    result = is_prefix("pre", "prefix")
    print('is_prefix("pre", "prefix")')
    print("  should return:  True")
    print("  yours returned: " + str(result))
```

**Calls the functions you implemented with test inputs**

**Prints expected vs. actual outputs**

**Only runs when runtests.py is run as a script, not imported!**

```
if __name__ == "__main__":
    args = get_command_line_args()
    if len(args) == 0:  # if there are no command-line arguments
        print("Please specify the test suite: pre, suf, after, replace, solve")
    else:
        which_question = args[0]  # reads the first command-line argument
        if which_question == "pre
            is_prefix_test1()
```

The rest of this code handles the command line arguments, e.g., `python3 runtests.py` **`pre`** and determines which functions are called