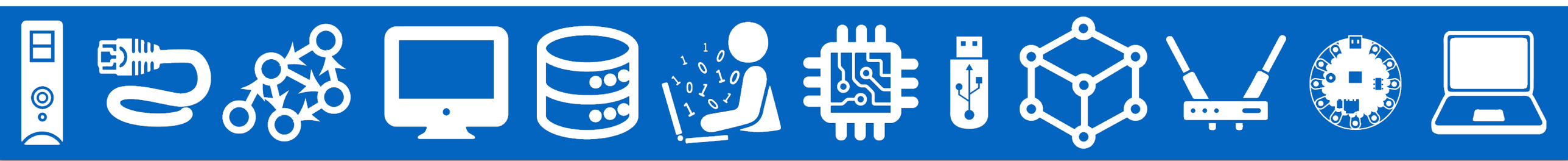
CS 134:
Nested Lists



Announcements & Logistics

- **HW 4** will be released today
- **Lab 4** due today/tomorrow
- **Lab 2 graded feedback**
  - Let us know if you have questions or concerns

Do You Have Any Questions?

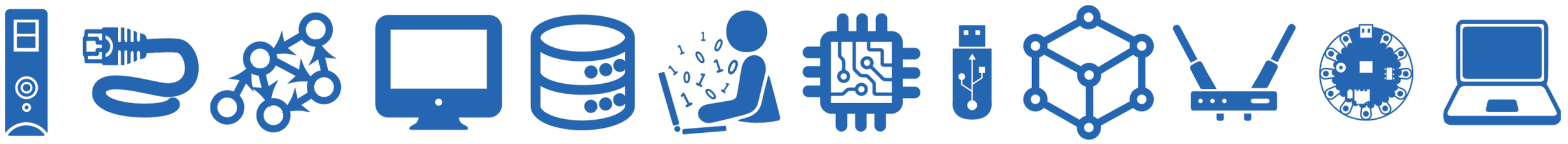
Last Time

- Introduced nested `for` loops
    - Discussed how to trace the execution of loops
    - Learn about the `range` sequence type
- Determined when/how to use **flag variables**

Today's Plan

- Introduce and use **nested lists**
- More examples of iteration:
    - Iterate over nested sequences and collect/filter useful statistics
- Module vs scripts
    - How to import and test functions
    - Role of the special `if __name__ == "__main__":` code block

Nested Lists



Nested Lists

- Remember, any object can be an element of a list.
  - This includes other lists!
- That is, we can have **lists of lists** (sometimes called a two-dimensional list)!
- Suppose we want to create a **list of lists of strings** called `my_lst`

```
>>> str_lst1 = ['cat']  # list of str
>>> str_lst2 = ['dog']  # list of str

>>> my_lst = [str_lst1] + [str_lst2]
>>> my_lst
[['cat'], ['dog']]
```

Nested Lists

```
                element
                   ↓
my_lst = [ ['cat', 'frog'],
           ['dog', 'toad'],  ←—— row
         ['cow', 'duck'] ]
```

- Suppose we want to access individual elements from our nested list
  - We can do so using the square bracket notation!

```
my_lst[row][element]  # evaluates to a str
```

  - `row` is index into "***outer***" list (identifies ***which inner list*** we want). In other words, defines the "row" you want.
  - `element` is index into "***inner***" list (identifies ***which element*** within the inner list). In other words, defines the "column" you want.

```
>>> my_lst[1][0]
'dog'
```

Lists and Data Types

- Python is a loosely typed programming language
  - We don't explicitly declare data types of variables
    - But every value still has a data type!
  - It's important to make sure we pay attention to what a function expects, especially with lists and strings! (remember this in Lab 4)

- **Lists of <u>lists</u> of strings** versus **list of strings**:

```
my_lst = [ ['cat', 'frog'],
           ['dog', 'toad'],
          ['cow', 'duck'] ]
```

```
my_lst = ['cat', 'frog',
          'dog', 'toad',
          'cow', 'duck']
```

```
my_lst[1][0] is 'dog'
```

```
my_lst[1][0] is 'f'
```

Sequence Operations

```
characters = [['Elizabeth Bennet', 'Fitzwilliam Darcy'],
              ['Harry Potter', 'Ron Weasley'],
              ['Frodo Baggins', 'Samwise Gamgee'],
              ['Julius Ceasar', 'Brutus']]

>>> len(characters)  # what is this?
4

>>> len(characters[0]) # what is this?
2

>>> characters = characters + ['Rhett Butler', 'Scarllet O Hara']
[['Elizabeth Bennet', 'Fitzwilliam Darcy'],
 ['Harry Potter', 'Ron Weasley'],
 ['Frodo Baggins', 'Samwise Gamgee'],
 ['Julius Ceasar', 'Brutus'],
 'Rhett Butler',
 'Scarllet O Hara']
```

Be careful when concatenating lists of two different types

Looping Over Nested Lists

```
characters =
[['Elizabeth Bennet', 'Fitzwilliam Darcy', 'Charles Bingley'],
['Harry Potter', 'Ron Weasley', 'Hermoine Granger'],
['Frodo Baggins', 'Samwise Gamgee', 'Gandalf']]
```

```
for char_list in characters:
    print(char_list)
    for name in char_list:
        print(name)
```

Loops over the "outer lists"

Prints each inner list one by one

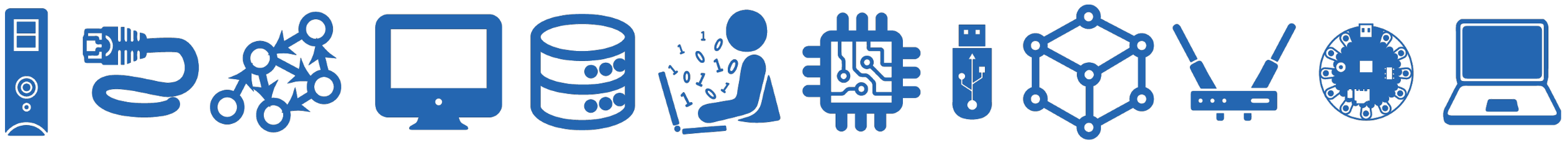
Prints each individual name one by one

Loops over the names in each "inner list"

Why Nested Lists?

- Nested Lists are useful to represent tabular data
    - Example: data stored in google sheets
- Each inner list is a row
- List of lists: collection of all rows (the whole table)
- Lets take an example of real data that we can store as list of lists

Querying Sequences



Querying Lists

- Asking for the min/max of a list according to a specified definition, or filtering the list according to some criteria is a task that comes up frequently in computer science.
- **ex: produce a list of names with the most number of vowels from a list of names**
- Decomposing the problem:
    - Will need to know if a character is a vowel or not
    - Will need to count the number of vowels
    - Will need to keep track of highest number
    - If something's higher, overwrite the old max

`is_vowel()` function

- Consider two versions of an `is_vowel()` function that takes a character (a string) as input and returns whether or not it is a vowel
  - Use `in` operator to simplify code (fewer boolean expressions)

```
def old_is_vowel(c):
    """ is_vowel function """
    return (c == 'a' or c == 'e' or c == 'i' or c == 'o' or c == 'u' or c == 'A' or c == 'E' or c == 'I' or c == 'O' or c == 'U')
```

```
def is_vowel(char):
    """ Simpler is_vowel function """
    return char in 'aeiouAEIOU'
```

Counting Vowels

- We can use a for loop to implement a `count_vowels()` function
- Notice how `count` "accumulates" values in the loop
- Recall, `count` here is called an **accumulation variable**

```
def count_vowels(word):
    """ Takes a string as input and returns
    the number of vowels in it """

    count = 0 # initialize the counter

    # iterate over the word one character at a time
    for char in word:
        if is_vowel(char): # call helper function
            count = count + 1
    return count
```

Exercise: Name Fun Facts!

- Write a function **`max_vowels`** that can be used to identify what the most number of vowels in all student names are. (Hint: use **`count_vowels()`** which returns the number of vowels in a string.)

```
def max_vowels(name_list):
    """ Takes a list of strings name_list and returns the number
    representing the maximum number of vowels in a name"""
```

- General strategy for finding max in list of sequences?
  - Initialize a max value BEFORE the loop to a very small number
  - If you see a value bigger than max while looping, update max

```
>>> max_vowels(["Lida", "Mark", "Rohit", "Anna", "Genevieve", "Maximilian"])
5
```

Exercise: Name Fun Facts!

- Write a function **`max_vowels`** that can be used to identify what the most number of vowels in all student names are. (Hint: use **`count_vowels()`** which returns the number of vowels in a string.)

```
def max_vowels(name_list):
    """ Takes a list of strings name_list and returns the number
    representing the maximum number of vowels in a name """

    max_so_far = 0

    for name in name_list:
        count = count_vowels(name)
        if count > max_so_far:
            # update found a name with more vowels
            max_so_far = count



    return max_so_far
```

What if we wanted a list of the names with the maximum number of vowels, instead of count?

```
>>> max_vowels(["Lida", "Mark", "Rohit", "Anna", "Genevieve", "Maximilian"])
5
```

Exercise: Name Fun Facts!

- Write a function **`most_vowels`** that can be used to compute the list of students with the most vowels in their first name. (Hint: use **`count_vowels()`** which returns the number of vowels in a string.)

```
def max_vowels(name_list):
    """ Takes a list of strings name_list and returns the number
    representing the maximum number of vowels in a name """

    max_so_far = 0

    for name in name_list:
        count = count_vowels(name)
        if count > max_so_far:
            # update found a name with more vowels
            max_so_far = count

    return max_so_far
```

Will need to initialize accumulator variable, and return that instead of the max num

New max found, throw out old accumulated values

What if it has the same number as our max? Add it to our accumulator!

```
>>> max_vowels(["Lida", "Mark", "Rohit", "Anna", "Genevieve", "Maximilian"])
5
```

Exercise: Name Fun Facts!

- Write a function **`most_vowels`** that can be used to compute the list of students with the most vowels in their first name. (Hint: use **`count_vowels()`** which returns the number of vowels in a string.)

```
def most_vowels(name_list):
    """ Takes a list of strings name_list and returns a list
    of names with the most number of vowels """

    max_so_far = 0
    result = []
    for name in name_list:
        count = count_vowels(name)
        if count > max_so_far:
            # update found a name with more vowels
            max_so_far = count
            result = [name]

        elif count == max_so_far:
            result = result + [name]

    return result
```

How would you modify this to compute the least number of vowels instead?

```
>>> most_vowels(["Lida", "Mark", "Rohit", "Anna", "Genevieve", "Maximilian"])
['Genevieve', 'Maximilian']
```

Exercise: Name Fun Facts!

- Write a function **`least_vowels`** that can be used to compute the list of students with the least vowels in their first name. (Hint: use **`count_vowels()`** again.)

```
def most_vowels(name_list):
    """ Takes a list of strings name_list and returns a list
    of names with the most number of vowels """

    max_so_far = 0
    result = []
    for name in name_list:
        count = count_vowels(name)
        if count > max_so_far:
            # update found a name with more vowels
            max_so_far = count
            result = [name]

        elif count == max_so_far:
            result = result + [name]

    return result
```

Rather than set the max low, set the min high?

Rather than looking for vals higher, want lower!

Need to use consistent variable names

```
>>> most_vowels(["Lida", "Mark", "Rohit", "Anna", "Genevieve", "Maximilian"])
['Genevieve', 'Maximilian']
```

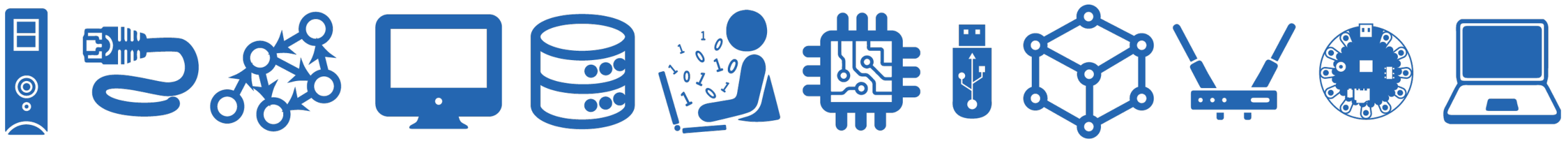
Exercise: Student Fun Facts!

- Write a function **`least_vowels`** that can be used to compute the list of students with the least vowels in their first name. (Hint: use **`count_vowels()`** again.)

```
def least_vowels(name_list):
    """ Takes a list of strings, name_list, and returns a list
    of names with the least number of vowels """

    min_so_far = 100000 # when might this break? Do we have something better?
    result = []
    for name in name_list:
        count = count_vowels(name)
        if count < min_so_far:
            # update found a name with fewer vowels
            min_so_far = count
            result = [name]

        elif count == min_so_far:
            result = result + [name]

    return result
```

```
>>> least_vowels(["Lida", "Iris", "Rohit", "Anna", "Genevieve", "Maximilian"])
['Lida', 'Iris', 'Rohit', 'Anna']
```

Nested Lists Additional Examples



Nested Loops and Nested Lists

- Let us trace through the code below:

```
def mystery2(lst_lsts):
    new_lstlsts = []
    for row in lst_lsts:
        new_row = []
        for item in row:
            new_row = new_row + [item*item]
        new_lstlsts = new_lstlsts + [new_row]
    return new_lstlsts

list_of_lists = [[1,2,3], [4,5,6], [7,8,9]]
print(mystery2(list_of_lists))
```

Nested Loops

- Trace through the code below:

new_lstlsts	row	new_row	item
`[]`	`[1,2,3]`	`[]`	

```
def mystery2(lst_lsts):
    new_lstlsts = []
    for row in lst_lsts:
        new_row = []
        for item in row:
            new_row = new_row + [item*item]
        new_lstlsts = new_lstlsts + [new_row]
    return new_lstlsts
```

```
lst_lsts = [[1,2,3],
            [4,5,6],
            [7,8,9]]
```

Nested Loops

- Trace through the code below:

new_lstlsts	row	new_row	item
[]	[1,2,3]	[]	
		[1]	1
	[1,2,3]	[1,4]	2
[[1,4,9]]	[1,2,3]	[1,4,9]	3

```
def mystery2(lst_lsts):
    new_lstlsts = []
    for row in lst_lsts:
        new_row = []
        for item in row:
            new_row = new_row + [item*item]
        new_lstlsts = new_lstlsts + [new_row]
    return new_lstlsts
```

```
lst_lsts = [[1,2,3],
            [4,5,6],
            [7,8,9]]
```

Nested Loops

- Trace through the code below:

new_lstlsts	row	new_row	item
[]	[1,2,3]	[]	
		[1]	1
	[1,2,3]	[1,4]	2
	[1,2,3]	[1,4,9]	3
[[1,4,9]]	[4,5,6]	[]	
		[16]	4
	[4,5,6]	[16,25]	5
[[1,4,9], [16,25,36]]	[4,5,6]	[16,25,36]	6

```
def mystery2(lst_lsts):
    new_lstlsts = []
    for row in lst_lsts:
        new_row = []
        for item in row:
            new_row = new_row + [item*item]
        new_lstlsts = new_lstlsts + [new_row]
    return new_lstlsts
```

```
lst_lsts = [[1,2,3],
            [4,5,6],
            [7,8,9]]
```

Nested Loops

- Trace through the code below:

new_lstlsts	row	new_row	item
[]	[1,2,3]	[]	
		[1]	1
	[1,2,3]	[1,4]	2
	[1,2,3]	[1,4,9]	3
[[1,4,9]]	[4,5,6]	[]	
		[16]	4
	[4,5,6]	[16,25]	5
[[1 ,4 ,9],	[4,5,6]	[16,25,36]	6
[16,25,36]]		[]	
[[1 ,4 ,9],	[7,8,9]	[49]	7
[16,25,36],		[49,64]	8
[49,64,81]]	[7,8,9]	[49,64,81]	9
	[7,8,9]		

```
def mystery2(lst_lsts):
    new_lstlsts = []
    for row in lst_lsts:
        new_row = []
        for item in row:
            new_row = new_row + [item*item]
        new_lstlsts = new_lstlsts + [new_row]
    return new_lstlsts
```

```
lst_lsts = [[1,2,3],
            [4,5,6],
            [7,8,9]]
```

Nested Loops

- Trace through the code below:

new_lstlsts	row	new_row	item
`[]`	`[1,2,3]`	`[]`	
		`[1]`	1
	`[1,2,3]`	`[1,4]`	2
	`[1,2,3]`	`[1,4,9]`	3
`[[1,4,9]]`	`[4,5,6]`	`[]`	
		`[16]`	4
	`[4,5,6]`	`[16,25]`	5
`[[1 ,4 ,9 ],`	`[4,5,6]`	`[16,25,36]`	6
`[16,25,36]]`		`[]`	
`[[1 ,4 ,9 ],`	`[7,8,9]`	`[49]`	7
`[16,25,36],`		`[49,64]`	8
`[49,64,81]]`	`[7,8,9]`	`[49,64,81]`	9
	`[7,8,9]`		

```
def mystery2(lst_lsts):
    new_lstlsts = []
    for row in lst_lsts:
        new_row = []
        for item in row:
            new_row = new_row + [item*item]
        new_lstlsts = new_lstlsts + [new_row]
    return new_lstlsts
```

```
lst_lsts = [[1,2,3],
            [4,5,6],
            [7,8,9]]
```

Nested Loops

```
def mystery2(lst_lsts):
    new_lstlsts = []
    for row in lst_lsts:
        new_row = []
        for item in row:
            new_row = new_row + [item*item]
        new_lstlsts = new_lstlsts + [new_row]
    return new_lstlsts
```

Accumulation variable

Accumulation variable

Note the [ ]

Why?!

```
list_of_lists = [[1,2,3], [4,5,6], [7,8,9]]
print(mystery2(list_of_lists))
```

Nested Loops

```
def mystery2(lst_lsts):
    new_lstlsts = []
    for row in lst_lsts:
        new_row = []
        for item in row:
            new_row = new_row + [item*item]
        new_lstlsts = new_lstlsts + [new_row]
    return new_lstlsts
```

Accumulation variable

Accumulation variable

Note the [ ]

Why?!

```
list_of_lists = [[1,2,3], [4,5,6], [7,8,9]]
print(mystery2(list_of_lists))
```

Why 2 accumulation variables?!

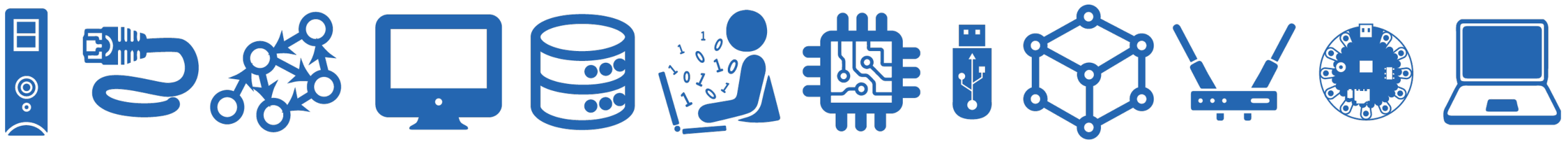
The square brackets ensure that we're adding a **`list to a list`**!

The inner loop accumulates the items for the row, the outer loop accumulates the rows

What would be a good function name for **`mystery2`**?

Something like **`power_table`**

Exercise: Nested Loops & 2 Lists of Lists simultaneously

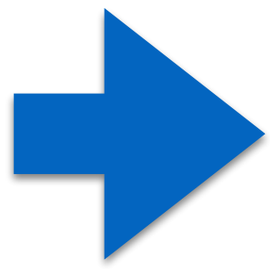


Previous Example: Nested Loops

- From last class:

```
def power_table(lst_lsts):
    new_lstlsts = []
    for row in lst_lsts:
        new_row = []
        for item in row:
            new_row = new_row + [item*item]
        new_lstlsts = new_lstlsts + [new_row]
    return new_lstlsts

list_of_lists = [[1,2,3], [4,5,6], [7,8,9]]
print(power_table(list_of_lists))
```

```
lst_lsts = [[1,2,3],
            [4,5,6],
            [7,8,9]]
```

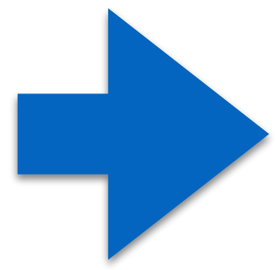


```
[[1,4 ,9],
[16,25,36],
[49,64,81]]
```

Nested Loops with Range

- What if instead of making a new list of lists of power, we wanted to **add** two different (equivalently sized) matrixes of numbers?
- e.g., instead of:

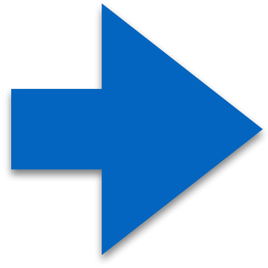
```
ll_b4 = [[1,2,3],
         [4,5,6],
         [7,8,9]]
```



```
ll_af = [[1 ,4 ,9 ],
         [16,25,36],
         [49,64,81]]
```

- We wanted:

```
ll1 = [[1,2,3],
       [4,5,6],
       [7,8,9]]
```



```
ll2 = [[7,8,9],
     + [1,1,1],
       [3,2,1]]
```

```
ll_af = [[8 ,10,12],
         [5 ,6 ,7 ],
         [10,10,10]]
```

Previous Example: Nested Loops

- From last class:

Will need 2 different lists of lists

```
def power_table(lst_lsts):
    new_lstlsts = []
    for row in lst_lsts:
        new_row = []
        for item in row:
            new_row = new_row + [item*item]
        new_lstlsts = new_lstlsts + [new_row]
    return new_lstlsts

list_of_lists = [[1,2,3], [4,5,6], [7,8,9]]
print(power_table(list_of_lists))
```

This needs to change, to add two items from separate tables

How do we look at items from two different lists of lists in a nested loop?

Can use range for parallel iteration!

```
def add_matrices(ll1, ll2):
    new_lstlsts = []
    for ri  in range(len(ll1)):
        new_row = []
        for ci in range(len(ll1[ri])):
            new_row = new_row + [ll1[ri][ci] + ll2[ri][ci]]
        new_lstlsts = new_lstlsts + [new_row]
    return new_lstlsts

list_of_lists1 = [[1,2,3], [4,5,6], [7,8,9]]
list_of_lists2 = [[7,8,9], [1,1,1], [3,2,1]]
print(add_matrices(list_of_lists1, list_of_lists2))
```

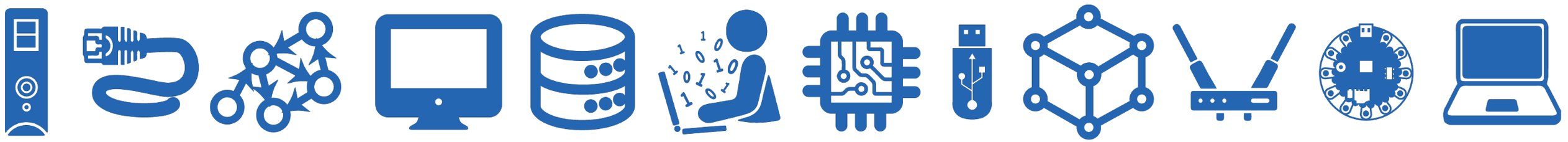
2 different lists of lists

Adds two items from separate tables

Determining range values from list of list values

- Add matrices, instead of squaring matrix

Modules & Scripts



Modules and Scripts: Example Code

leap.py

```
def is_leap(year):
    """Takes a year (int) as input and returns
    True if it is a leap year, else returns False"""

    # if not divisible by 4, return False
    if year % 4 != 0:
        return False

    # is divisible by 4 but not divisible by 100
    # return True
    elif year % 100 != 0:
        return True

    # is divisible by 4 and divisible by 100
    # but not divisible by 400, return False
    elif year % 400 != 0:
        return False

    # is divisible by 400 (and also 4, and 100)
    # return True
    return True
```

Modules and Scripts

- A **script** is a piece of code saved in a file, e.g., `leap.py`
  - Meant to be executed with: `python3 leap.py`
- A **module** is a collection of function definitions saved in a file (like a script)
  - Meant to be imported and used by other scripts
  - Can be used in interactive python
- Code in a `.py` file can serve as both a module and a script
- To distinguish between these two modes of operation, we can check the value of the special variable called `__name__`
- *Note: If a variable starts/ends with double __ in Python, it's a special variable*

Modules and Scripts

- Consider the code we wrote in `leap.py`
- When `leap.py` is run as a **script** then the `__name__` variable is set to the string `"__main__"`
- When we import the code as a **module**, the `__name__` variable is set to the name of the module `leap`

- Why does this matter?
    - We often want different behavior when the code is run as a script vs when it's imported as a module

`if __name__ == '__main__':`

- This is just an if statement with an equality Boolean expression:
  - Checks whether the `__name__` variable is set to the string `'__main__'`. Tells us the code is being run as a script
- We place code that we want to run only when our module is executed as a script inside the `if __name__ == "__main__"`: block
- Useful for testing code that we do not want to run when we import functions in interactive Python

Example: Script vs Module

```
# name.py
# test the role of __name__variable
print("__name__ is set to", __name__, "\n\n")
```

```
terminal % python3 name.py
__name__ is set to __main__

terminal % python 3
Python 3.10.8 (main)
Type "help", "copyright", "credits" or "license" for more information.
>>> import name
__ name__ is set to name
```

leap.py

```python
# function to check if a given year is a leap year

def is_leap(year):
    """Takes a year (int) as input and returns
    True if it is a leap year, else returns False"""

    # if not divisible by 4, return False
    if year % 4 != 0:
        return False

    # is divisible by 4 but not divisible by 100
    # return True
    elif year % 100 != 0:
        return True

    # is divisible by 4 and divisible by 100
    # but not divisible by 400, return False
    elif year % 400 != 0:
        return False

    # is divisible by 400 (and also 4, and 100)
    # return True
    return True


#  following code only run when run as a script
if __name__ == "__main__":
    # ask user to enter year
    year = int(input("Enter a year: "))

    # call isLeap
    if is_leap(year):
        print(year, "is a leap year!")
    else:
        print(year, "is not a leap year.")
```

Running leap as a Script and Module

Running leap as a Script and Module

- Running leap.py as a script (notice the code in the if block runs!)

```
terminal$ python3 leap.py
Enter a year: 1900
1900 is not a leap year.
terminal$ python3 leap.py
Enter a year: 2040
2040 is a leap year!
```

- Running leap.py as a module in interactive Python

```
terminal$ python3
Python 3.10.8 (main)
Type "help", "copyrigh...
>>> from leap import *
>>> is_leap(1900)
False
>>> is_leap(2040)
True
```

The end!