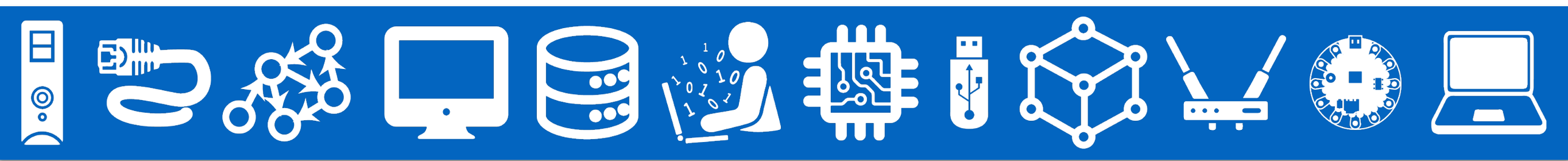
CS134:
Lists and Loops



Announcements & Logistics

- **Lab 3** released today
  - Builds upon everything we've learned so far (including Monday's content):
    - Iterating over sequences (strings, lists, ranges) as well as conditionals
  - More "moving pieces" than Lab 2
  - *Please come* to help hours if you have questions (or to say hi!)
    - **Prelab** due at the beginning of lab
- **HW 3** due Monday at 10 pm on Gradescope

Do You Have Any Questions?

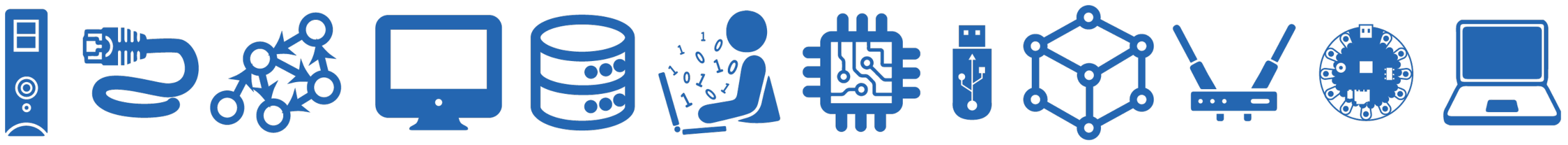
Last Time

- Introduce iteration using **for loops** to iterate over **sequences**
- Discussed sequence indexing using [ ] and using the `len()` function
    - And slicing [:]
    - And stepping [::2]
    - And negative indices!
    - And `in` operator!

Today's Plan

- Introduce a new data type (which is also a sequence):
  - `list`
- Learn more about sequences
  - **`in`** operator
  - sequence "slicing"
- Iterating over and "accumulating" using lists

Lists

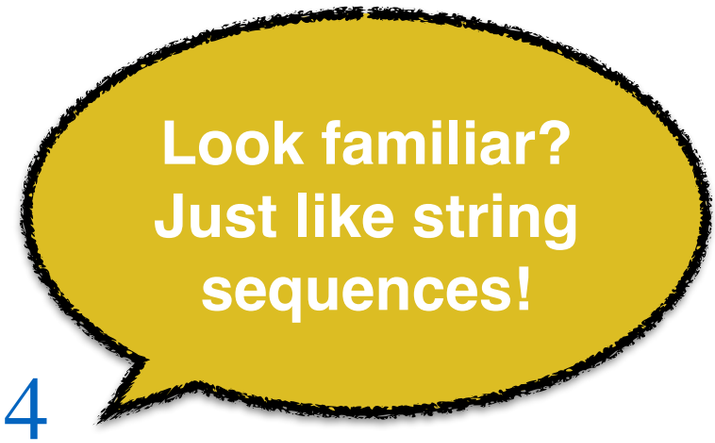


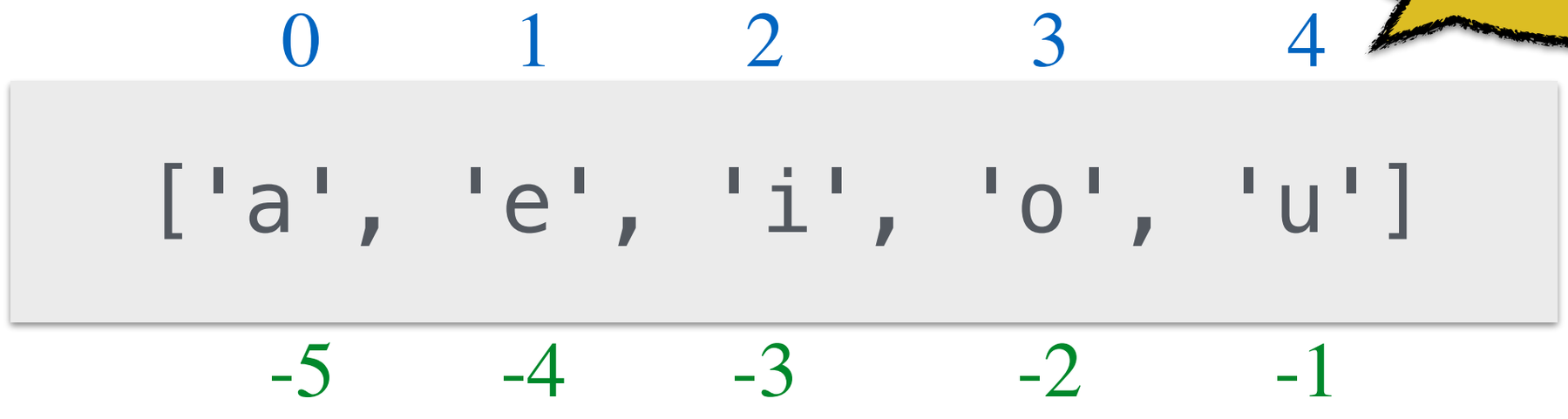
A New Sequence: Lists

- A list is a comma separated, ordered sequence of values.
- These values can be **heterogenous** (strings, ints, floats, etc)
    - Example: `my_list = ['Hello', 42, 23.5, True]`
    - Remember, we **zero-index**! So we say that `'Hello'` is at **index** 0, `42` is at **index** 1, and so on
- Like strings, we can access each element of a list using these **indices**

How Do Indices Work?

- Can access elements of a sequence (such as a list) using its **index**
- Indices in Python are both positive and negative
- Everything outside of these values will cause an `IndexError`.






```
vowels = ['a', 'e', 'i', 'o', 'u']
```

Features of Lists

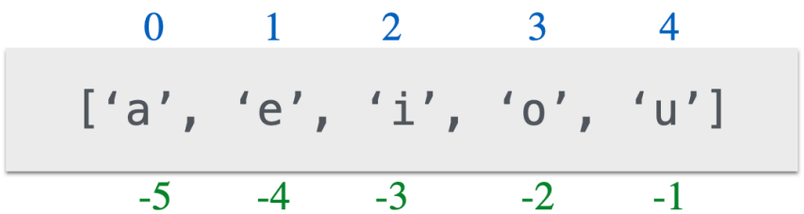
- Lists are:
  - **Comma separated, ordered sequences** of values
  - Can be **heterogenous**: multiple types can appear in the same list
  - **Mutable** (or "changeable") objects in Pythons. In contrast, strings are **immutable** (they cannot be changed).
    - We will discuss ***mutability*** in more detail soon!

```
# Examples of various lists:
>>> word_lst = ["What", "a", "beautiful", "day"]
>>> num_lst = [1, 5, 8, 9, 15, 27]
>>> char_lst = ['a', 'e', 'i', 'o', 'u']
>>> mixed_lst = [3.14, 'e', 13, True]
>>> type(num_lst)
list
```

Lists can be heterogeneous (mixed)!

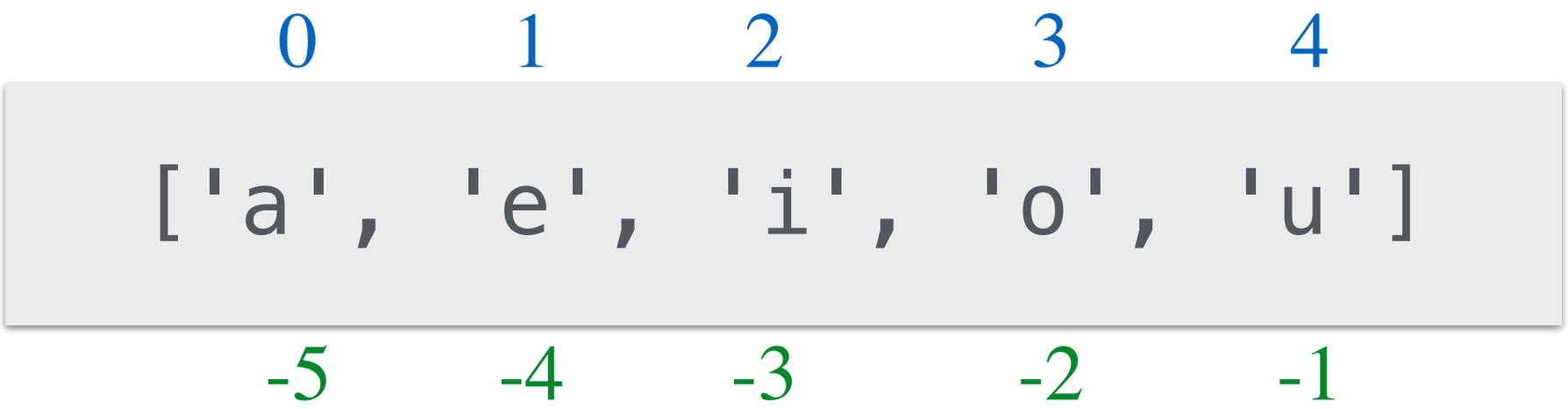
Accessing Elements of Sequences

```
>>> vowels = ['a', 'e', 'i', 'o', 'u']
>>> vowels[0] # character at 0th index?
'a'
>>> vowels[3] # character at 3rd index?
'o'
>>> vowels[4] # character at 4th index?
'u'
>>> vowels[5] # will this work?
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```



Negative Indexing

- Negative indexing starts from -1, and provides a handy way to access the last character of a non-empty sequence without knowing its length




```
>>> vowels = ['a', 'e', 'i', 'o', 'u']
>>> vowels[-1]
'u'
```

Note: Most other languages do not support negative indexing!

Slicing Sequences

- We can extract **sub**sequences of a sequence using the **slicing** operator `[:]`
- For a given sequence `var`, `var[start:end]` returns a new sequence starting at index '`start`' (inclusive), ending at index '`end`' (exclusive)
- Example: Suppose we want to extract the sublist `['a','e']` from `vowels` using slicing operator `[:]`

```
>>> vowels = ['a', 'e', 'i', 'o', 'u']
>>> # return the sequence from 0th index up to 1st
>>> # (not including 2nd)
>>> vowels[0:2]
['a','e']
```

Slicing Sequences: Using Step

- The (optional) third `step` parameter to the slicing operator determines in what direction to traverse, and whether to skip any elements while traversing and creating the subsequence
- By default, `start = 0, end = len(), step = +1` (which means move left to right in increments of one)
- If we omit any of the three parameters, slice uses the default values

```
>>> evens = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
>>> evens[0:5] # start is 0, end is 5, step is +1
 [2, 4, 6, 8, 10]
>>> evens[:8:2] # start is 0, end is 8, step is +2
 [2, 6, 10, 14]
>>> evens[::2] # start is 0, end is 10, step is +2
 [2, 6, 10, 14, 18]
```

Slicing Sequences: Optional Step

- When the step parameter is set to a negative value it gives a nifty way to reverse sequences
- Note: `start` and `end` are interpreted "backwards" when using a negative step!

```
>>> evens = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
>>> evens[::-1] # reverse the sequence
[20, 18, 16, 14, 12, 10, 8, 6, 4, 2]
>>> evens[::-2]
[20, 16, 12, 8, 4]
>>> evens[8:0:-1]
[18, 16, 14, 12, 10, 8, 6, 4]
```

Sequences in Python: Strings

- **Sequences** in Python represent **ordered collections of elements**: e.g., strings, lists, ranges, etc.
- A **`string`** is an ordered sequences of individual characters
  - Example: `word = "Hello"`
- A **`list`** is a comma-separated, ordered sequence of values
  - Example: `num_list = [1, 5, 8, 9, 15, 27]`
- In CS, we use **zero-indexing**, so we say that `'H'` is at **index** 0 of `word`, `8` is at **index** 2 of `num_list`, and so on
- We can access each character of a sequence using **indices**

```
>>> word[1]
'e'
```

```
>>> num_list[4]
15
```

Slicing Sequences

- We can **extract subsequences** of a sequence using the **slicing** operator `[:]`
- For a given sequence `var`,

  `var[start:end]`

  returns a new sequence of the same type that contains the elements starting at index '`start`' (*inclusive*) and ending at index '`end`' (*exclusive*)

```
>>> vowels = 'aeiou'
>>> vowels[0:2]
'ae'
>>> num_list = [2, 4, 8, 16]
>>> num_lst = [0:-1] # everything except last
[2, 4, 8]
```

Slicing Sequences

- We can **extract subsequences** of a sequence using the **slicing** operator `[:]`
- For a given sequence `var`,

  `var[start:end:step]`

  returns a new sequence of the same type that contains the elements starting at index '`start`' (*inclusive*), ending at index '`end`' (*exclusive*), and using an **(optional)** increment of '`step`'

- By default (if not specified):
  - start defaults to 0 (the beginning of string)
  - end defaults to `len(var)` (end of string)
  - step defaults to `+1`

Examples

- **Question**. How would we reverse a sequence using slicing?

```
>>> evens = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
>>> evens[0:5]
[2, 4, 6, 8, 10]
>>> evens[:8:2]
[2, 6, 10, 14]
>>> evens[::2]
[2, 6, 10, 14, 18]

>>> name = "Ephelia"
>>> name[::-1]
'ailehpE
```

Testing Membership: `in` Operator

- The `in` operator in Python is used to test if a given sequence is a subsequence of another sequence; returns `True` or `False`

```
>>> "Williams" in "Williamstown"
True
```

```
>>> "w" in "Williams" # capitalization matters
False
```

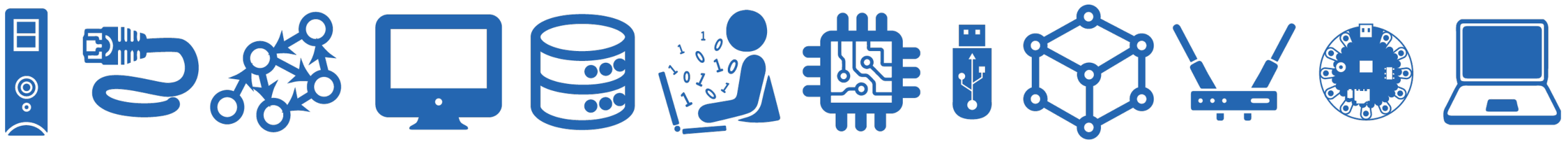
```
>>> dog_lst = ["Wally", "Velma", "Pixel", "Linus"]
>>> "Linus" in dog_lst
True
>>> “Artie" in dog_lst
False
```

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**!

Other List Operators



Length of a Sequence

- Python has a built-in `len()` function that computes the length of a sequence such as a list (or any other sequence like a string)
- For a list, `len()` returns the number of elements in the list
- Thus, any list called `words` has the following (positive) indices

`0, 1, 2, ..., len(words)-1`

```
>>> len(['a', 'e', 'i', 'o', 'u'])
5

>>> len(["Chels", "Artie", "Pixel", "Linus"])
4
```

Testing Membership: `in` Operator

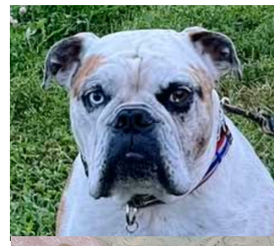
- The `in` operator in Python is used to test if a given sequence is a subsequence of another sequence; returns `True` or `False`

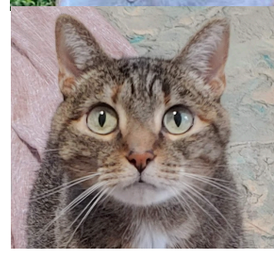
```
>>> "i" in ['a', 'e', 'i', 'o', 'u']
True
>>> "a" in ['a', 'e', 'i', 'o', 'u']
True
>>> "A" in ['a', 'e', 'i', 'o', 'u'] # caps matter
False
```

Membership in Sequences

- The `in` operator in Python is used to test if a given sequence is a subsequence of another sequence; returns True or False

```
>>> dog_lst = ["Chels", "Artie", "Pixel", "Linus"]
>>> "Linus" in dog_lst
True
>>> "Dizzy" in dog_lst
False
```



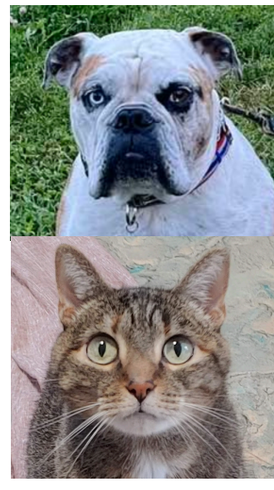


`not in` sequence operator

- The **`not in`** operator in Python returns True if and only if the given element is **not** in the sequence

```
>>> dog_lst = ["Chels", "Artie", "Pixel", "Linus"]
>>> "Linus" in dog_lst
True
>>> "Dizzy" in dog_lst
False

>>> "Dizzy" not in dog_lst
True
>>> "z" not in "Linus"
True
```



Note that **`not in`** also works for strings

List Concatenation

- We can use the **+** operator to ***concatenate*** lists together
- Creates a **new list** with the combined elements of the sublists
  - *Does not modify original lists!*

returns a **new list** with elements from aList and bList

```
>>> a_lst = ["the", "quick", "brown", "fox"]
>>> b_lst = ["jumped", "over", "the", "dogs"]
>>> a_lst + b_lst # concatenate lists
['the', 'quick', 'brown', 'fox', 'jumped', 'over', 'the', 'dogs']
>>> a_lst
['the', 'quick', 'brown', 'fox']
>>> b_lst = b_lst + ["back"] # add "back" to b_lst
>>> b_lst # since we reassign result to b_lst, b_lst has changed
['jumped', 'over', 'the', 'dogs', 'back']
```

a_lst is unchanged!

To change b_lst, we have to reassign b_lst to the new list

Review: Basic Operations on Sequences

```
>>> word_lst = ["What", "a", "beautiful", "day"]
>>> word_lst[3]
'day'

>>> word_lst[-1]
'day'
>>> len(word_lst)
4

>>> dog_lst = ["Chels", "Artie", "Pixel", "Linus"]
>>> dog_lst[2:4]
['Pixel', 'Linus']
```

Indexing lists using [ ]

Finding length of list using len( )

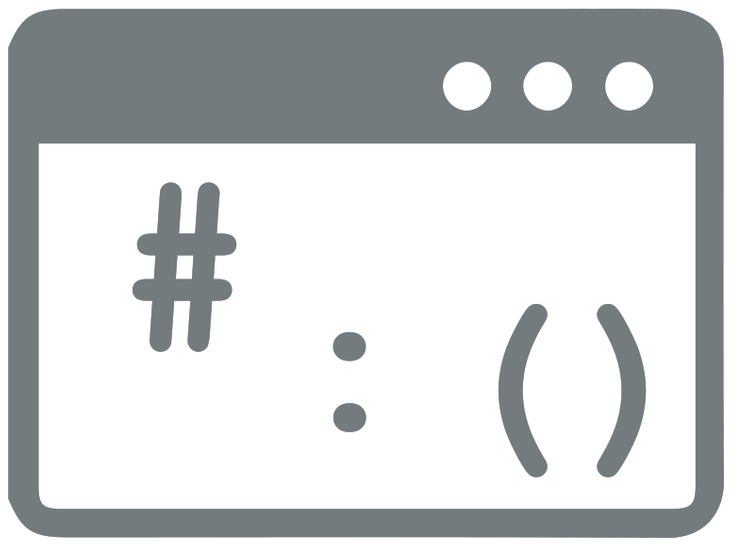
Slicing lists using [:] (can also use optional step)

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**!

Exercise: Palindromes



Exercise: `palindromes`

- A palindrome is a string that is the same forwards and backwards
- The following strings are all examples of palindromes:
  - `""` (any string with length 0)
  - `"x"` (any string with length 1)
  - `"aba"`
  - `"racecar"`
- The following strings are not palindromes:
- `"aA"` (Case mismatch)
- `"12321 "` (Un-matched space " " at end of string)

Exercise: `palindromes`

- Write a function that iterates over a given list of strings `s_list`, returns a (new) list containing all the strings in `s_list` that are the same forward and backwards (ignoring case).

```
>>> palindromes(["anna", "banana", "kayak", "rigor", "tacit", "hope"])
['anna', 'kayak']
>>> palindromes(["1313", "1110111", "0101"])
['1110111']
>>> palindromes(["level", "stick", "gag"])
['level', 'gag']
```

Exercise: `palindromes`

What is our high level algorithm, in words?

- Go through each word in `s_list`. If the word is a palindrome, append it to our "solution list". After reaching the end of our list, our "solution list" should contain all of the palindromes.

`lst = lst + [item]`

"accumulator" variable

`for` loop

conditional

function that returns a boolean

Solution: `palindromes`

```
def get_palindromes(s_list):
    '''Takes a list of string s_list and returns a new list
     containing strings from s_list that
     are the same forwards and backwards'''

    solution = [] # initialize the accumulation variable

    # iterate over each item in seq
    for item in s_list:
        # check if it's a palindrome (use the "range" version)
        if is_palindrome(item):
            # append the palindrome string to accumulation list
            solution = solution + [item]

    # return what we accumulated
    return solution
```

How do we implement `is_palindrome(word)`?

`is_palindrome(word)`

What is our high level algorithm, in words?

- Multiple correct algorithms exist!
  - Return true if word is equal to a reversed copy of word

```
def is_palindrome(word) :
    # use the "slicing trick" to reverse a string
    return word == word[::-1]
```

Loops: Take-Aways

- `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

The end!