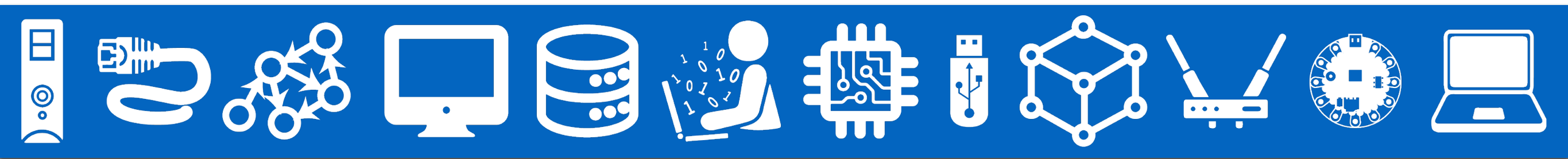
CS134:
List Comprehensions & Tuples



Announcements & Logistics

- **No Lab this week!**
- **HW 11** due tonight at 10 pm (on Gradescope)

- **Happy Thanksgiving!!**

Do You Have Any Questions?

Last Time

- We discussed two sorting algorithms
  - ***Selection sort***
    - Simple-to-implement sorting algorithm, $O(n^2)$
  - ***Merge sort***
    - An optimal divide-and-conquer sorting algorithm, $O(n \log_2 n)$

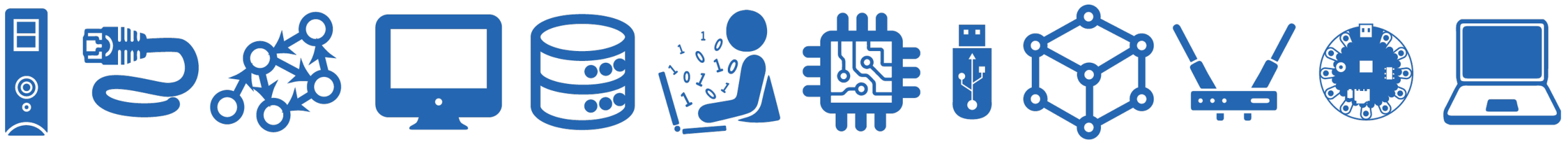
Today's Plan

- Introduce some nifty *pythonic* ways of doing things we've done previously:

- **List comprehensions**: a quick way to make simple lists!
- **Tuples:** an *immutable* sequence that's convenient for swapping values, etc

- Two more lectures after break:
    - Comparison of Python vs Java
    - OOP Wrap up and review

List Comprehensions



List Patterns: Map & Filter

- When using lists and loops, there are common patterns that appear
- **Mapping:** Iterate over a list and return a new list that results from ***performing an operation on each element*** of original list
    - E.g., take a list of integers `num_lst` and return a new list which contains the square of each number in `num_lst`
- **Filtering:** Iterate over a list and return a new list that results from ***keeping only elements of the original list that satisfy some condition***
    - E.g., take a list of integers `num_lst` and return a new list which contains only the even numbers in `num_lst`
- Python allows us to implement these patterns succinctly using **list comprehensions**

List Comprehensions

Mapping List Comprehension (perform operation on each element)

```
new_lst = [expression for item in sequence]
```

Filtering List Comprehension (only keep some elements)

```
new_lst = [item for item in sequence if conditional]
```

- Important points:
  - List comprehensions always start with an **expression** (even a variable name like "`item`" is an expression!)
  - We **never use += or .append()** inside of list comprehensions
  - We can **combine mapping and filtering** into a single list comprehension:

"**Combo" Comprehension** (perform operation on some elements)

```
new_lst = [expression for item in sequence if conditional]
```

Dissecting List Comprehensions

```
new_lst = [expression for item in sequence if conditional]
```

Task: Extract even numbers from a range and create a list of their squares.

```
result = []
for n in range(10):
    if n%2 == 0:
        result += [n**2]
```

Using a list comprehension:

```
result = [n**2 for n in range(10) if n%2 == 0]
```

expression — item — sequence — conditional

All list comprehensions can be rewritten using a for loop!

Using List Comprehensions

- **List comprehensions** are convenient when converting between lists of <types>:
- Recall our **`Book`** class from before:

```
class Book:
    def __init__(self, title):
        self._title = title
    def __str__():
        return "'" + self._title + "'"
```

- *Example*: How to convert **`book_lst`** from a list of **`Book`**s to a list of **`str`**ing titles?

```
>>> book_lst = [Book("LOTR"), Book("ParOfSower"), Book("Emma")]
```

Using List Comprehensions

- **List comprehensions** are convenient when converting between lists of <types>:
- Recall our **`Book`** class from before:

```
class Book:
    def __init__(self, title):
        self._title = title
    def __str__():
        return "'" + self._title + "'"
```

- *Example*: How to convert **`book_lst`** from a list of **`Book`**s to a list of **`str`**ing titles?

```
>>> book_lst = [Book("LOTR"), Book("ParOfSower"), Book("Emma")]
```

```
    b_strs = []
    for book in book_lst:
        b_strs += [str(book)]
```

sequence

item

expression

Using List Comprehensions

- **List comprehensions** are convenient when converting between lists of <types>:
- Recall our **`Book`** class from before:

```
class Book:
    def __init__(self, title):
        self._title = title
    def __str__():
        return "'" + self._title + "'"
```

- *Example*: How to convert **`book_lst`** from a list of **`Book`**s to a list of **`str`**ing titles?

```
>>> book_lst = [Book("LOTR"), Book("ParOfSower"), Book("Emma")]
```

expression item sequence

```
>>> [str(book) for book in book_lst]
['LOTR', 'ParOfSower', 'Emma']
```

Using List Comprehensions

- **List comprehensions** are convenient when working with files:
- Recall our `superheroes.csv` from before:

```
Wonderwoman,Strength,5
Superman,Strength,13
Spiderman,Spidey things,9
Black Panther,Technology,4
Captain Marvel,Strength,4
Starfire,Strength,1
Cyborg,Technology,1
Batman,Justice,23
Robin,Justice,2
Ms. Marvel,Light,0
Jean Grey,Telekenesis,7
Ironman,Technology,9
Forge,Technology,1
```

```
with open("data/superheroes.csv") as roster:
    lines = []
    for line in roster:
        lines += [line.strip()]
```

- Can now become:

```
with open("data/superheroes.csv") as roster:
    lines = [line.strip() for line in roster]
```

- Maybe we only want just the superpowers?

Using List Comprehensions

- **List comprehensions** are convenient when working with files:
- Recall our `superheroes.csv` from before:

```
Wonderwoman,Strength,5
Superman,Strength,13
Spiderman,Spidey things,9
Black Panther,Technology,4
Captain Marvel,Strength,4
Starfire,Strength,1
Cyborg,Technology,1
Batman,Justice,23
Robin,Justice,2
Ms. Marvel,Light,0
Jean Grey,Telekenesis,7
Ironman,Technology,9
Forge,Technology,1
```

- Maybe we only want just the superpowers:

```
with open("data/superheroes.csv") as roster:
    powers = []
    for line in roster:
        powers += [line.strip().split(',')[1]]
```

sequence

item

expression

```
with open("data/superheroes.csv") as roster:
    powers = [line.strip().split(',')[1] for line in roster]
```

sequence

expression

item

- Maybe we only want only the superheroes with "man" in their name?

Using List Comprehensions

- **List comprehensions** are convenient when working with files:
- Recall our `superheroes.csv` from before:

```
Wonderwoman,Strength,5
Superman,Strength,13
Spiderman,Spidey things,9
Black Panther,Technology,4
Captain Marvel,Strength,4
Starfire,Strength,1
Cyborg,Technology,1
Batman,Justice,23
Robin,Justice,2
Ms. Marvel,Light,0
Jean Grey,Telekenesis,7
Ironman,Technology,9
Forge,Technology,
```

- Maybe we only want only the superheroes with "man" in their name:

```
def get_name(line):
    return line.strip().split(',')[0]
```

```
with open("data/superheroes.csv") as roster:
    old_skool = []
    for line in roster:
        if "man" in get_name(line):
            old_skool += [get_name(line)]
```

sequence

expression

item

conditional

```
with open("data/superheroes.csv") as roster:
    old_sk = [get_name(line) for line in roster if "man" in get_name(line)]
```

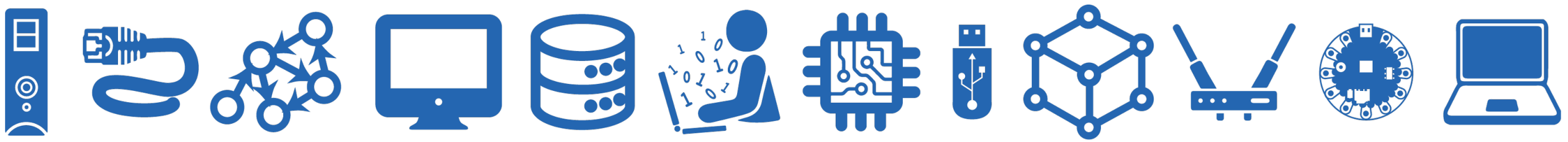
expression

item

sequence

conditional

Tuples



Tuples: An Immutable Sequence

- Tuples are an **immutable sequence of values** (almost like immutable lists) separated by commas and enclosed within parentheses **( )**

```
# string tuple
>>> names = ("Mark", "Iris", "Lida")

# int tuple
>>> primes = (2, 3, 5, 7, 11)

# singleton
>>> num = (5, )

# parentheses are optional
>>> values = 5, 6

# empty tuple
>>> emp = ()
```

A tuple of size 1 is called a singleton. Note the (funky) syntax.

Tuples as Immutable Sequences

- Tuples, like strings, support any sequence operation that ***does not involve mutation***: e.g,
  - `len()` function: returns number of elements in tuple
  - `[]` indexing: access specific element
  - `+, *`: tuple concatenation
  - `[:]`: slicing to return subset of tuple (as a new tuple)
  - `in` and `not in`: check membership
  - `for loop`: iterate over elements in tuple

Multiple Assignment and Unpacking

- Tuples support a simple syntax for assigning multiple values at once, and also for "unpacking" sequence values

```
>>> a, b = 4, 7

# reverse the order of values in tuple

>>> b, a = a, b

# tuple assignment to “unpack” list elements

>>> cb_info = ['Charlie Brown', 8, False]

>>> name, age, glasses = cb_info
```

- Note that the preceding line is just a more concise way of writing:

```
>>> name = cb_info[0]

>>> age = cb_info[1]

>>> glasses = cb_info[2]
```

Multiple Return from Functions

- Tuples come in handy when returning multiple values from functions

```
# multiple return values as a tuple
def arithmetic(num1, num2):
    '''Takes two numbers and returns the sum and product'''
    return num1 + num2, num1 * num2
```

```
>>> arithmetic(10, 2)
(12, 20)
>>> type(arithmetic(3, 4))
<class 'tuple'>
```

Conversion between Sequences

- The functions `tuple()`, `list()`, and `str()` convert between sequences

```
>>> word = "Williamstown"
>>> char_lst = list(word) # string to list
>>> char_lst
['W', 'i', 'l', 'l', 'i', 'a', 'm', 's', 't', 'o', 'w', 'n']
>>> char_tuple = tuple(char_lst) # list to tuple
>>> char_tuple
('W', 'i', 'l', 'l', 'i', 'a', 'm', 's', 't', 'o', 'w', 'n')
>>> list((1, 2, 3, 4, 5)) # tuple to list
[1, 2, 3, 4, 5]
```

Conversion between Sequences

- The functions `tuple()`, `list()`, and `str()` let us convert between sequences

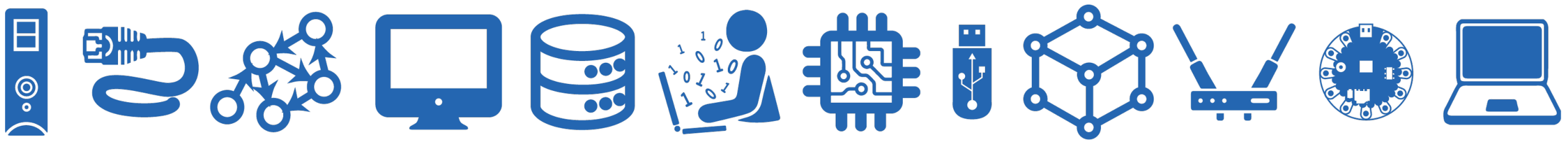
- The functions `tuple()`, `list()`, and `str()` convert between sequences


```
>>> str(('hello', 'world')) # tuple to string
"('hello', 'world')"
>>> num_range = range(12)
>>> list(num_range) # range to list
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
>>> str(list(num_range)) # range to list to string
'[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]'
```

Other Uses for Tuples?

- Why would we want a data type that looks and acts like a list but is less flexible?
- Being immutable can be helpful!
  - Tuples can be stored in sets
  - Tuples can be used as the keys for dictionaries
  - If we are representing data that should not change, using a tuple prevents accidental violations of our program's invariants
    - BoggleCube faces represent fixed physical objects
    - A mouseclick corresponds to a fixed point on the window
    - etc.

Useful String Methods



Useful String Methods

Discover more str methods with `pydoc3 str` !

```
>>> s = "     CSCI 134 is great!\n \t"
>>> s.strip()
'CSCI 134 is great!'
```

Remove whitespace from left/right sides of the `str`ing `s`

```
>>> lst = ['starry', 'starry', 'night']
>>> stars = '**'.join(lst)
>>> stars
'starry**starry**night'
```

Joins all elements from list of `str`, `lst`, using the leading `str '**'`

```
>>> stars.split('**')
['starry', 'starry', 'night']
```

Splits all elements from `str stars`, using the `str` argument `'**'`

```
>>> "I have {} {} & {} {}".format(2,'cats',1,'dog')
'I have 2 cats & 1 dog.'
```

Inserts arguments into the `{}` in the `str` instance object.

Other useful Tricks

- Handy Python "tricks"
  - Optional print() arguments
  - Other comprehensions are possible (sets and dicsts!)
  - sys.arg
- Handy Command-line "tricks"
  - Pipes (|)
  - Redirects (> and >>)
  - Background/foreground
  -

Takeaways

- Python has lots of built in features for doing things efficiently
  - But it's worthwhile to know how to build many of these features ourselves!
- **Tuples** are an *immutable* sequence type that:
  - supports all sequence operations such as indexing and slicing
  - is useful for argument unpacking, multiple assignments
  - is useful as a list-like data type without aliasing issues
- We **can** change the value of *mutable* objects such as lists
  - You can create a "true" copy of a list using slicing or a list comprehension

```
new_lst = my_lst[:]
new_lst = [ele for ele in my_lst]
```

The end!