# Welcome to Data C88C!

**Lecture 09: Sequences**
Monday, July 7th, 2025
Week 3
Summer 2025
Instructor: Eric Kim (ekim555@berkeley.edu)

# Announcements

- (Optional, extra credit) Weekly course surveys on Gradescope
  - Out now: "Course Survey (Week 02) (optional, extra credit)"
  - Help make the class better, for both this semester and future semesters!
- Due dates:
  - Lab03, HW03 due date extended: now due tonight (July 7th, 11:59 PM PST) [link]
  - Lab04, HW04: due Tues July 8th
- Project 01 "Maps" released tomorrow
  - Group size: 2 (max)
  - Please try to find a partner, but if you'd prefer to work alone, that's fine too

# (Reminder) Announcement: Financial Aid Eligibility Survey

"In accordance with federal requirements established by the Department of Education, we need to verify that students are participating in their courses. A survey has been sent to your students in DATA C88C, COMPSCI C88C to confirm their eligibility to receive financial aid. Students will receive separate instructions to complete the 1-question assignment on academic integrity.

You can learn more about the requirement on the Eligibility for Financial Aid at UC Berkeley page."

**Students**: please check your bCourses for an assignment that verifies your participation in classes. **Required for receiving financial aid**. Read the above link for more info.

# Lecture Overview

- Sequences
  - Lists, str
- Range
- List comprehensions
- Slicing

# Definition: Sequence [Docs]

- In Python: the term **sequence** refers generally to a data structure consisting of an indexed collection of values, which we'll generally call elements.
  - That is, there is a first, second, third value (which CS types call #0, #1, #2, etc.). "Zero-based" vs "One-based" indexing
- A sequence may be finite (with a length) or infinite.
- It may be **mutable** (elements can change) or **immutable**.
- It may be **indexable**: its elements may be accessed via selection by their indices.
- It may be **iterable**: its values may be accessed sequentially from first to last.

```
# list
>>> my_nums = [42, 1, 5]

# indexing
>>> my_nums[0]
42

# modifying the list
>>> my_nums[2] = 3
>>> my_nums
[42, 1, 3]

# iterating through the list
>>> some_nums = [1, 2, 3, 4]
>>> for num in some_nums:
...     print(num * 2)
...
2
4
6
8
```

## Common sequence operations for this course

- `seq[ind]`: Indexing. Retrieval element at index `ind`
  - Note: Python uses zero-based indices! seq[0], seq[1], ...
- `len(seq)`: returns the length (or size) of the input sequence
- `elem in seq` / `elem not in seq`: check if `elem` is present in a sequence
- `seq1 + seq2`: concatenate two input sequences together (creates a new sequence)
-

# Iterating through a sequence

- Two common ways to iterate through a sequence: `for` and `while` loops

```
my_nums = [1, 2, 3]
for num in my_nums:
    print(num * 2)
```

```
my_nums = [1, 2, 3]
i = 0
while i < len(my_nums):
    print(my_nums[i] * 2)
    i += 1
```

# Sequence concatenation

- We can concatenate (aka fuse/join) two sequences via the `+` operator

```
>>> nums1 = [1, 2, 3]
>>> nums2 = [4, 5, 6]
>>> nums3 = nums1 + nums2
>>> nums3
[1, 2, 3, 4, 5, 6]
```

## Sequence "contains" element" (in, not in)

- We can check if a sequence contains an element

```
>>> nums1 = [1, 2, 3, 4]
>>> 2 in nums1
True
>>> 5 not in nums1
True
```

# Sequence types in Python

- Sequences that we've seen (or will see) in this course
  - string
    - Note: strings are immutable!
  - range
  - list
  - tuple
    - aka an immutable list

# Sequence example: strings

- Strings are an immutable sequence
- All of the sequence operations can also be performed on strings!

```
words = ['apple', 'ymca']
for word in words:
    for letter in word:
        print(letter + '!')
a!
p!
p!
l!
e!
y!
m!
c!
a!
```

iterate through `words` list

iterate through each letter (character) of the current word

add (concatenate) a "!" to the end of the letter

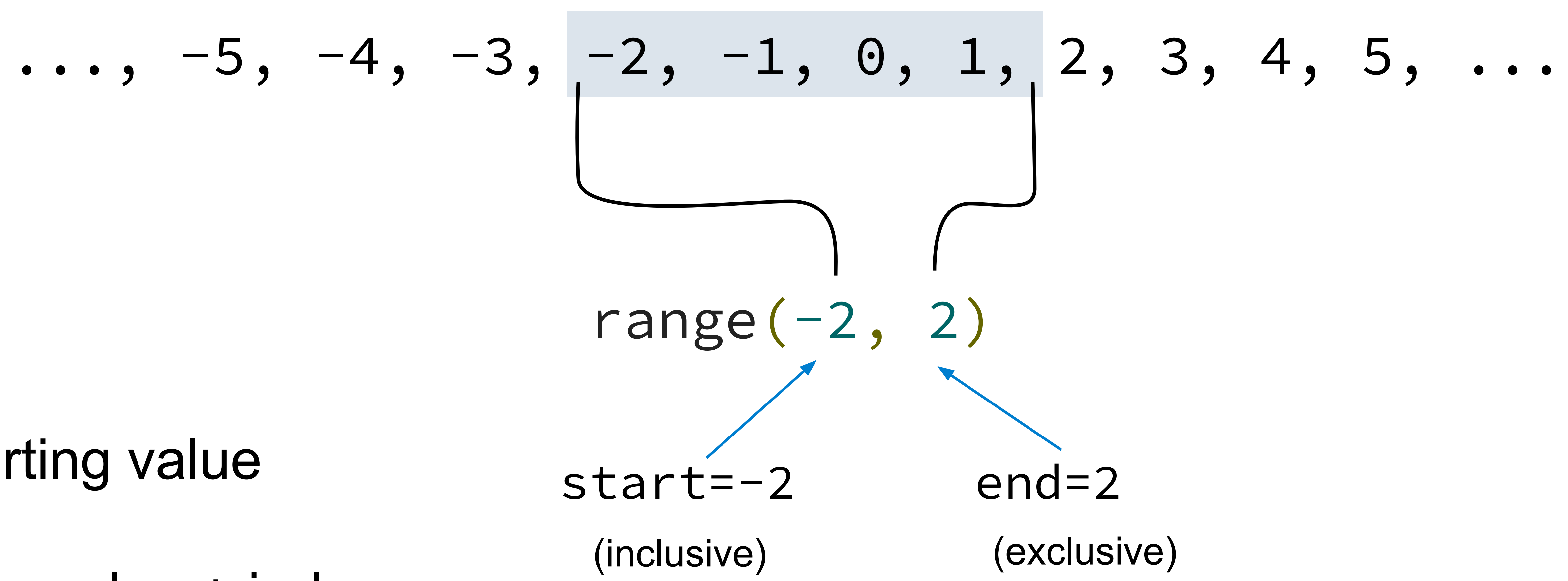
# (for reference) Sequence operations

| Operation | Result |
|---|---|
| x in s | True if an item of s is equal to x, else False |
| x not in s | False if an item of s is equal to x, else True |
| s + t | the concatenation of s and t |
| s * n or n * s | equivalent to adding (concatenating) s to itself n times |
| s[i] | ith item of s, origin 0 |
| s[i:j] | slice of s from i to j |
| s[i:j:k] | slice of s from i to j with step k |
| len(s) | length of s |
| min(s) | smallest item of s |
| max(s) | largest item of s |
| s.index(x[, i[, j]]) | index of the first occurrence of x in s (at or after index i and before index j) |
| s.count(x) | total number of occurrences of x in s |

# Ranges

# The Range Type

A range is a sequence of consecutive integers.*



**Length**: ending value - starting value

**Element selection**: starting value + index

```
>>> list(range(-2, 2))
[-2, -1, 0, 1]
```

List constructor

```
>>> list(range(4))
[0, 1, 2, 3]
```

Range with a 0 starting value

**Tip**: `range()` does not immediately calculate the entire sequence at once. Instead, it generates each element "on demand" (called "lazy evaluation"). To fully materialize the range, one way is to use the `list()` constructor.

```
>>> range(5)
range(0, 5)
>>> list(range(5))
[0, 1, 2, 3, 4]
>>> [x for x in range(5)]
[0, 1, 2, 3, 4]
```

(Demo 09.py:Demo00)

* Ranges can actually represent more general integer sequences.

# List Comprehensions

# List Comprehensions

[<map exp> for <name> in <iter exp> if <filter exp>]

Short version: [<map exp> for <name> in <iter exp>]

A way of turning a simple for loop into a single line
(not 100% accurate, but one way of looking at it):

```
my_nums = [1, 2, 3, 4]
out = []
for num in my_nums:
    if is_even(num):
        out = out + [num ** 2]
print(out)
# [4, 16]
```

```
my_nums = [1, 2, 3]
out = [num ** 2 for num in my_nums if is_even(num)]
print(out)
# [4, 16]
```

# Example: Two Lists

Given these two related lists of the same length:

```
xs = range(-10, 11)
ys = [x*x - 2*x + 1 for x in xs]
```

**Question**: Write a list comprehension that evaluates to:

A list of all the x values (from xs) for which the corresponding y (from ys) is below 10.

**Question (practice)**: implement this with a for loop, and a while loop

```
xs_where_y_is_below_10 = []
for i in range(len(xs)):
    if ys[i] < 10:
        xs_where_y_is_below_10 += [xs[i]]

i = 0
xs_where_y_is_below_10 = []
while i < len(xs):
    if ys[i] < 10:
        xs_where_y_is_below_10 += [xs[i]]
    i += 1
```

```
>>> list(xs)
[-10,  -9, -8, -7, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4,  5,  6,  7,  8,  9, 10]
>>> ys
[121, 100, 81, 64, 49, 36, 25, 16,  9,  4, 1, 0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> xs_where_y_is_below_10
[-2, -1, 0, 1, 2, 3, 4]
```

**Answer**:

```
[xs[i] for i in range(len(xs)) if ys[i] < 10]
```

# Example: Promoted

# First in Line

Implement **promoted**, which takes a sequence **s** and a one-argument function **f**. It returns a list with the same elements as **s**, but with all elements **e** for which **f(e)** is a true value ordered first. Among those placed first and those placed after, the order stays the same.

```
def promoted(s, f):
    """Return a list with the same elements as s, but with all
    elements e for which f(e) is a true value placed first.

    >>> promoted(range(10), odd)  # odds in front
    [1, 3, 5, 7, 9, 0, 2, 4, 6, 8]
    """
             [e for e in s if f(e)] + [e for e in s if not f(e)]
    return ____________________________________________________
```

# Lists, Slices, & Recursion

# A List is a First Element and the Rest of the List

For any list **s**, the expression **s[1:]** is called a *slice* from index 1 to the end (or 1 onward)

- The value of s[1:] is a list whose length is one less than the length of s
- It contains all of the elements of s except s[0]
- Slicing s doesn't affect s (it creates a **new** list)

```
>>> s = [2, 3, 6, 4]
>>> s[1:]
[3, 6, 4]
>>> s
[2, 3, 6, 4]
```

In a list **s**, the first element is **s[0]** and the rest of the elements are **s[1:]**.

# More slicing/indexing tricks

- Tip: negative indices generally means "count backwards from the end"

| Operation | Result |
|---|---|
| seq[start:end:step] | Slice a sequence from [start, end), but with stepsize=step.<br>Omitting start implicitly sets start=0<br>Omitting end implicitly sets end=len(seq).<br>Omitting stepsize implicitly sets stepsize=1. |
| seq[::-1] | Creates a new seq in reverse order. |
| seq[-k] | Return the element at index `len(seq) - k`, aka count backwards from the end. `seq[-1]` is "the last element", `seq[-2]` is "the second-to-last element", etc. |

# Recursion Example: Sum

Implement **sum_list**, which takes a list of numbers s and returns their sum. If a list is empty, the sum of its elements is 0.

```
def sum_list(s):
    """Sum the elements of list s.

    >>> sum([2, 4, 1, 3])
    10
    """

    if len(s) == 0:

        return 0

    else:
                s[0]        sum_list(s[1:])
        return _______ + _________________
```

**Recursive idea**: The sum of the elements of a list is the result of adding the first element to the sum of the rest of the elements

# Recursion Example: Large Sums

**Definition:** A sublist of a list s is a list with some (or none or all) of the elements of s.

Implement **large**, which takes a list of positive numbers **s** and a non-negative number **n**.

It returns the sublist of **s** with the largest sum that is less than or equal to **n**.

You may call **sum_list**, which takes a list and returns the sum of its elements.

```
def large(s, n):
    """Return the sublist of positive numbers s with the
    largest sum that is less than or equal to n.

    >>> large([4, 2, 5, 6, 7], 3)
    [2]  # 2 <= 3
    >>> large([4, 2, 5, 6, 7], 8)
    [2, 6]  # 2 + 6 = 8 <= 8
    >>> large([4, 2, 5, 6, 7], 19)
    [4, 2, 6, 7]  # 4 + 2 + 6 + 7 = 19 <= 19
    >>> large([4, 2, 5, 6, 7], 20)
    [2, 5, 6, 7]  # 2 + 5 + 6 + 7 = 20 <= 20
    """
    if s == []:
        return []
    elif s[0] > n:
        return large(s[1:], n)
    else:
        first = s[0]
                    [first] + large(s[1:], n - first)
        with_s0 = ______________________________________
                              large(s[1:], n)
        without_s0 = ___________________________________
        if sum_list(with_s0) > sum_list(without_s0):
            return with_s0
        else:
            return without_s0
```

# Alternate implementation: Large Sums

**Definition:** A sublist of a list s is a list with some (or none or all) of the elements of s.

Implement **large**, which takes a list of positive numbers **s** and a non-negative number **n**.

It returns the sublist of **s** with the largest sum that is less than or equal to **n**.

**Question**: why don't I have to check if `sum_without_s0 <= n` here?

eg why isn't it this?

```
...
elif sum_without_s0 <= n:
    return without_s0
```

**Answer**: the recursive call `without_s0 = large_v2(s[1:], n)` already enforces that the sum of `without_s0` is <= n.
"Trust in the recursion", and think about function domain + range.

Note: you can add the check and it would still work, it would just be redundant.

```
def large_v2(s, n):
    """Return the sublist of positive numbers s with the largest sum up
to n.

    >>> large_v2([4, 2, 5, 6, 7], 20)
    [2, 5, 6, 7]
    """
    # Alternate recursive implementation
    if s == []:
        return []
    elif n < 0:
        # s contains only positive integers, and it's
        # impossible to add pos ints to get a neg/zero int
        return []
    else:
        first = s[0]  # a number
        with_s0 = [first] + large_v2(s[1:], n - first)
        without_s0 = large_v2(s[1:], n)
        sum_with_s0 = sum_list(with_s0)
        sum_without_s0 = sum_list(without_s0)
        if sum_with_s0 > sum_without_s0 and sum_with_s0 <= n:
            return with_s0
        else:
            return without_s0
```