Programming Principles in Python (CSCI 503/490)

Sequences & Functions

Dr. David Koop

(some slides adapted from Dr. Reva Freedman)

Sequences

- Strings `"abcde"`, Lists `[1, 2, 3, 4, 5]`, and Tuples `(1, 2, 3, 4, 5)`

- Defining a list: `my_list = [0, 1, 2, 3, 4]`
- But lists can store different types:
  - `my_list = [0, "a", 1.34]`
- Including other lists:
  - `my_list = [0, "a", 1.34, [1, 2, 3]]`
- Others are similar: tuples use parenthesis, strings are delineated by quotes (single or double)

Sequence Operations

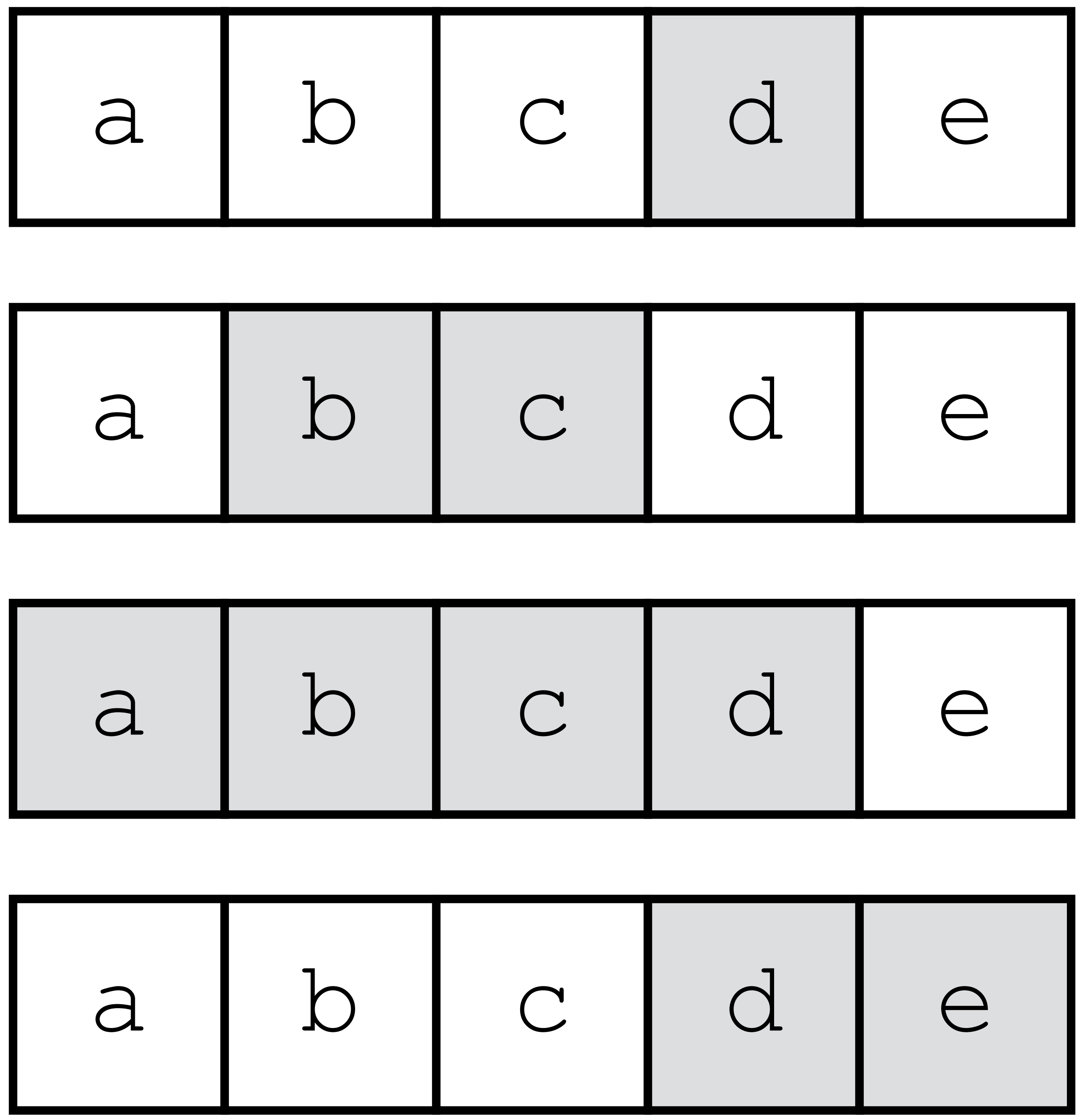
- Concatenate: `[1, 2] + [3, 4] # [1,2,3,4]`
- Repeat: `[1,2] * 3 # [1,2,1,2,1,2]`
- Length: `my_list = [1,2]; len(my_list) # 2`

- Concatenate: `(1, 2) + (3, 4) # (1,2,3,4)`
- Repeat: `(1,2) * 3 # (1,2,1,2,1,2)`
- Length: `my_tuple = (1,2); len(my_tuple) # 2`

- Concatenate: `"ab" + "cd" # "abcd"`
- Repeat: `"ab" * 3 # "ababab"`
- Length: `my_str = "ab"; len(my_str) # 2`

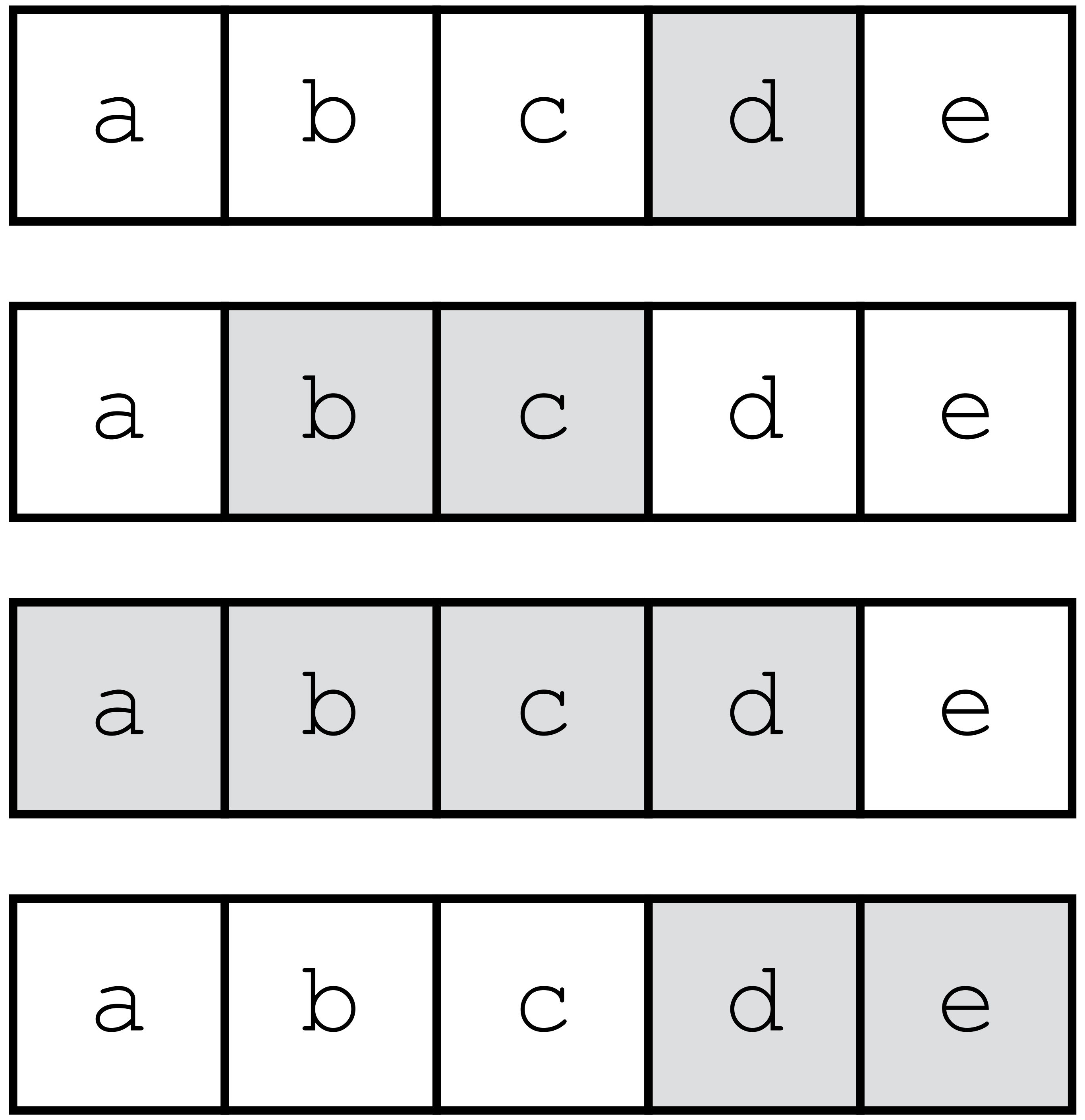
Indexing & Slicing Quiz

```
my_list = ['a', 'b', 'c', 'd', 'e']
```



Indexing & Slicing Quiz

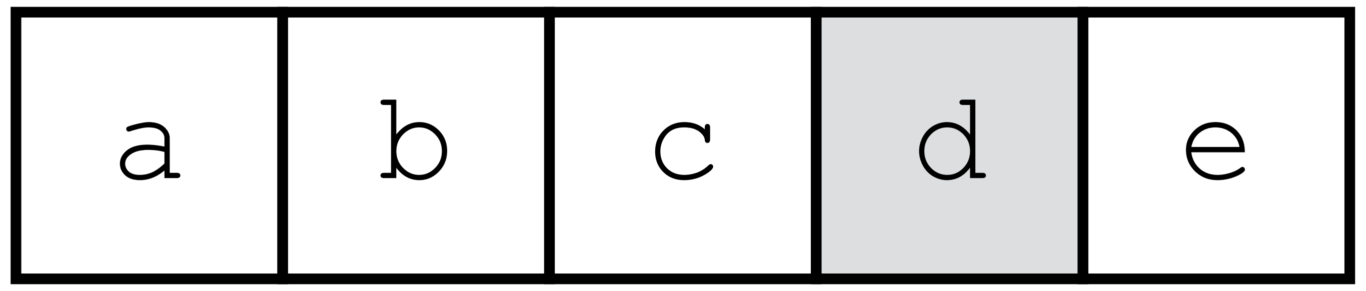
```
my_list = ['a', 'b', 'c', 'd', 'e']
```



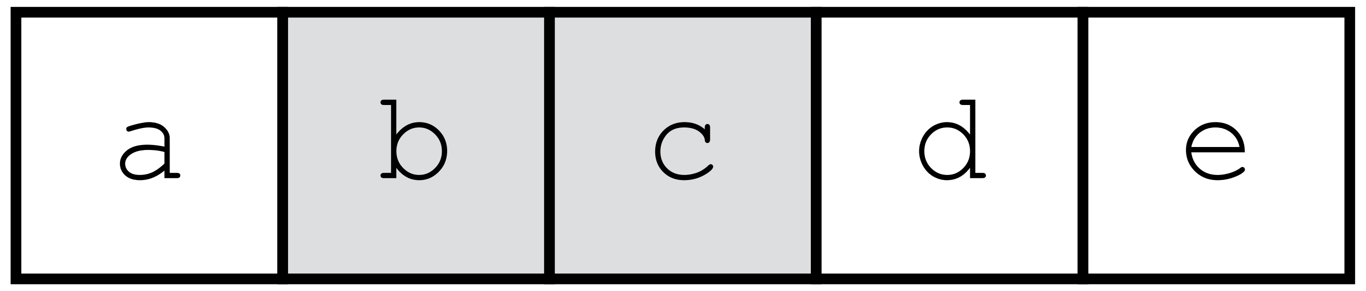
```
my_list[3]; my_list[-2]; my_list[3:4]
```

Indexing & Slicing Quiz

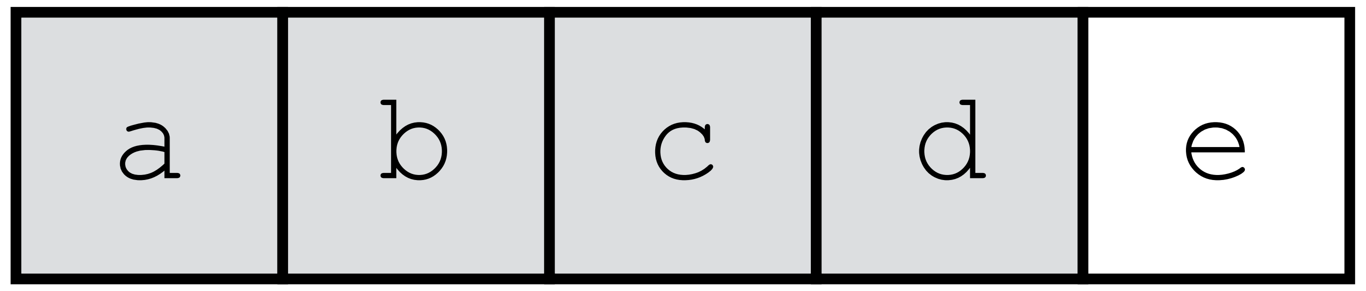
```
my_list = ['a', 'b', 'c', 'd', 'e']
```

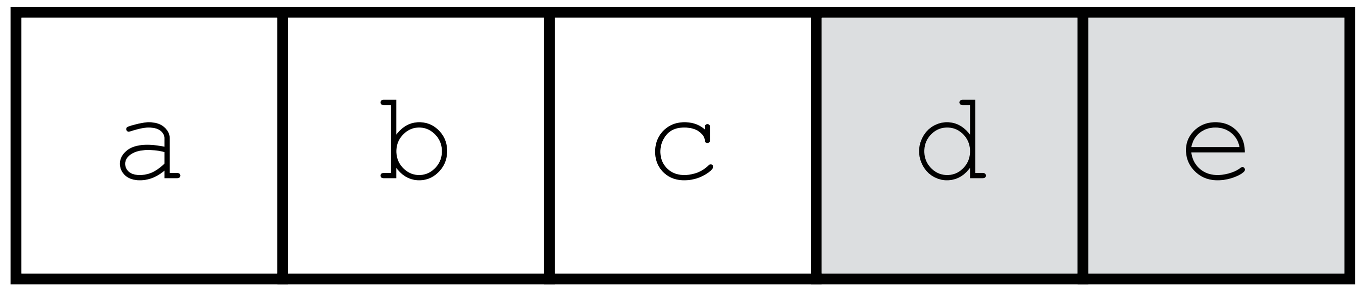


```
my_list[3]; my_list[-2]; my_list[3:4]
```



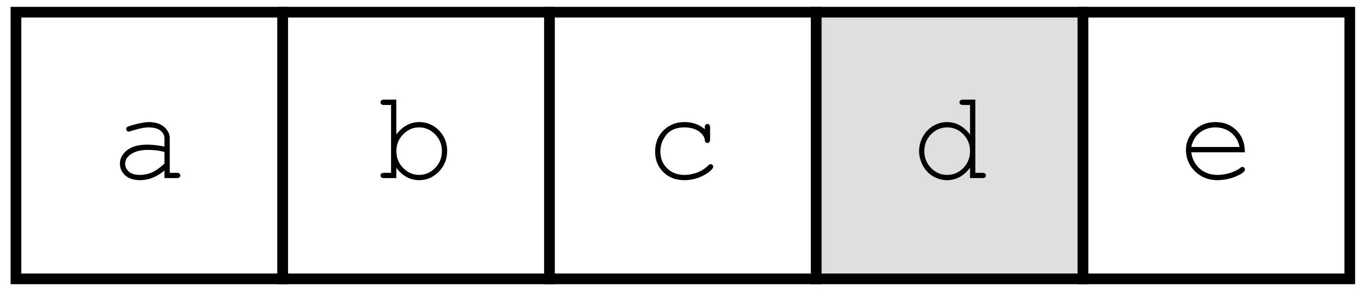
```
my_list[1:3]; my_list[-4:-2];
my_list[1:-2]
```



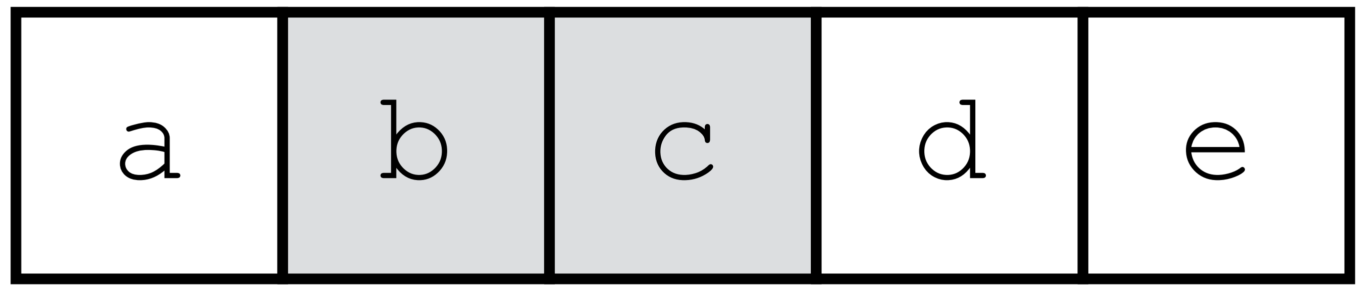


Indexing & Slicing Quiz

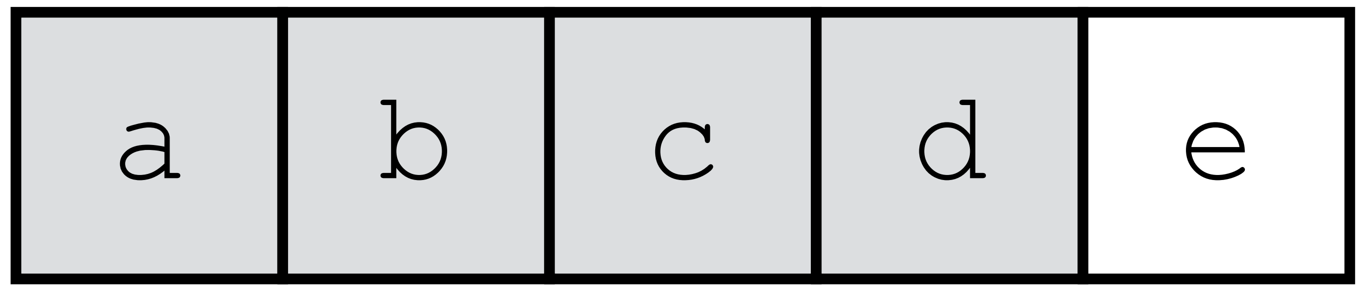
```
my_list = ['a', 'b', 'c', 'd', 'e']
```



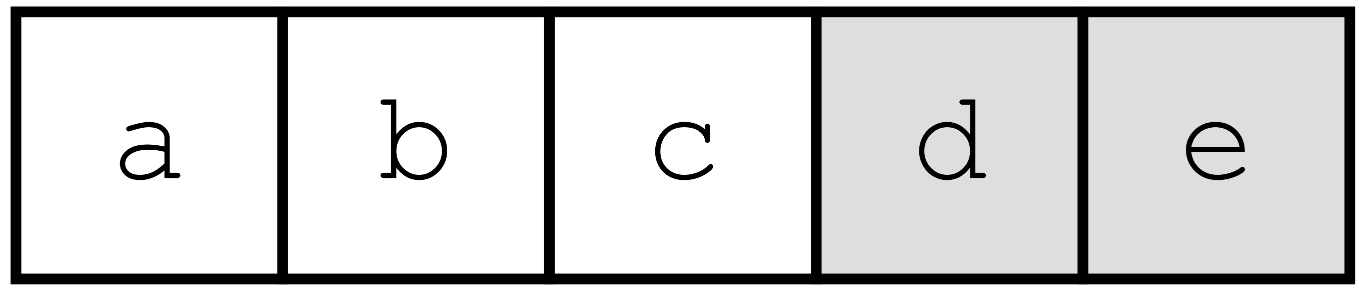
```
my_list[3]; my_list[-2]; my_list[3:4]
```



```
my_list[1:3]; my_list[-4:-2];
my_list[1:-2]
```

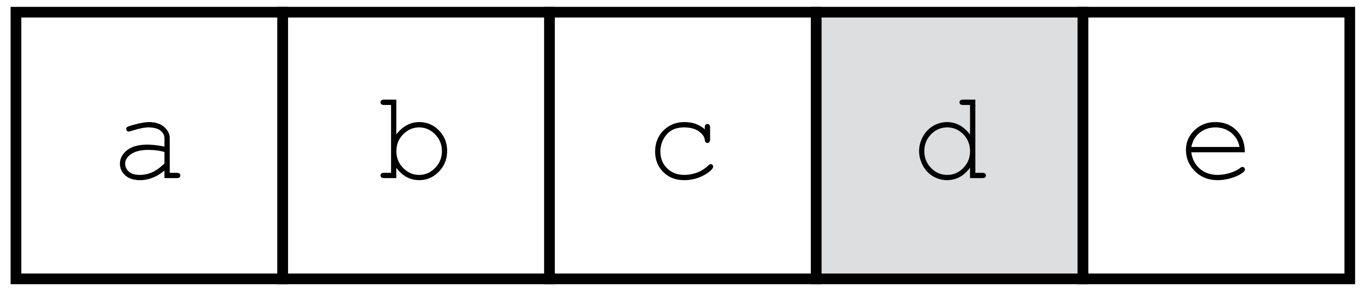


```
my_list[0:4]; my_list[:4];
my_list[-5:-1]
```

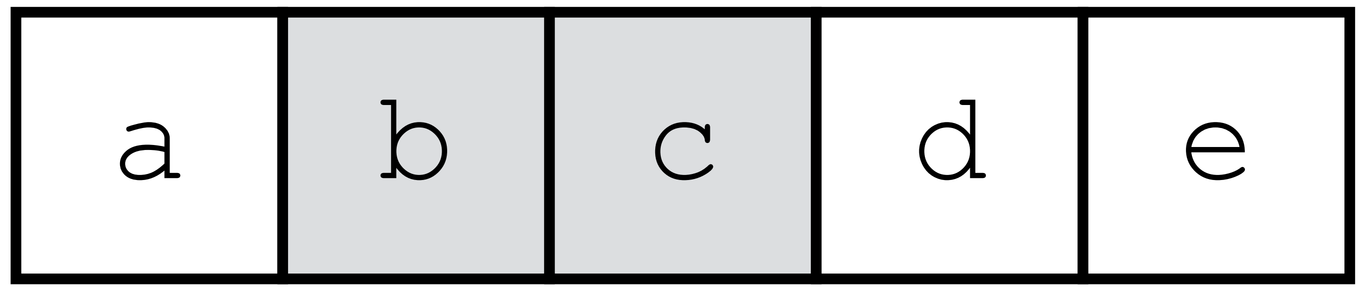


Indexing & Slicing Quiz

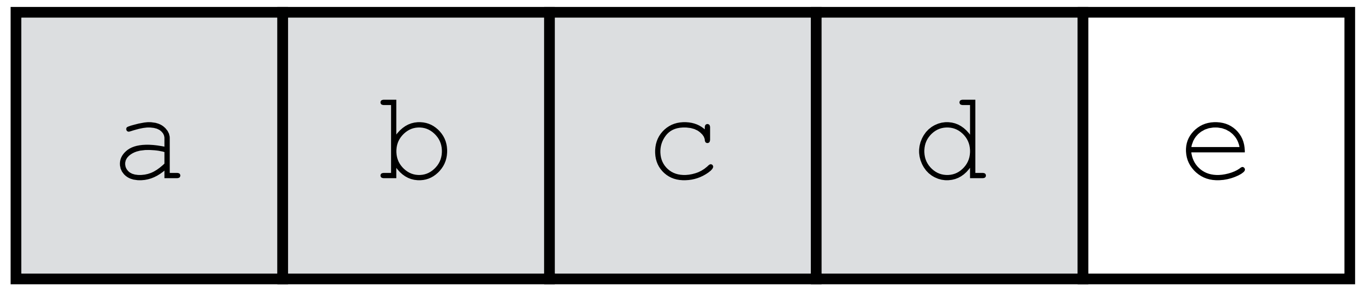
```
my_list = ['a', 'b', 'c', 'd', 'e']
```



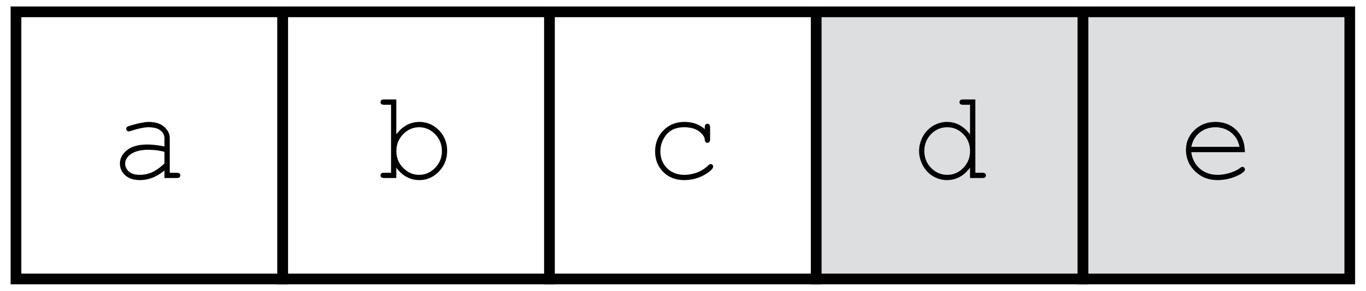
```
my_list[3]; my_list[-2]; my_list[3:4]
```



```
my_list[1:3]; my_list[-4:-2];
my_list[1:-2]
```



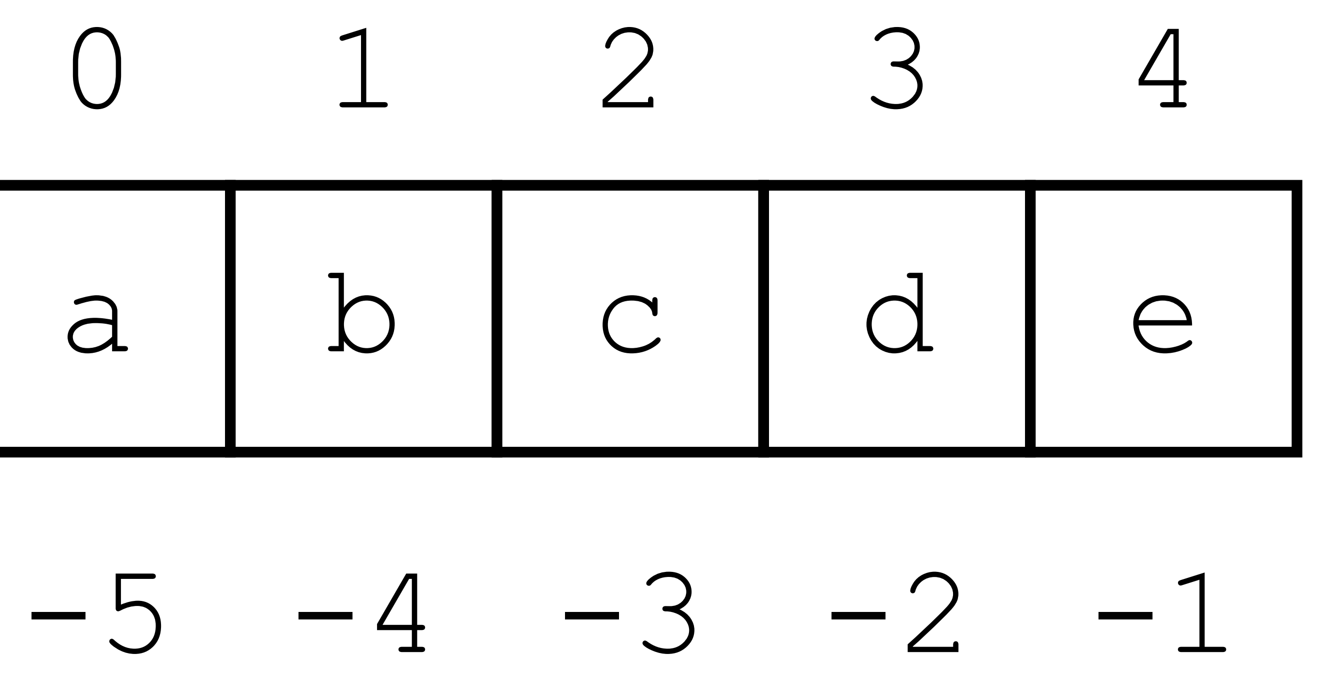
```
my_list[0:4]; my_list[:4];
my_list[-5:-1]
```



```
my_list[3:]; my_list[-2:]
```

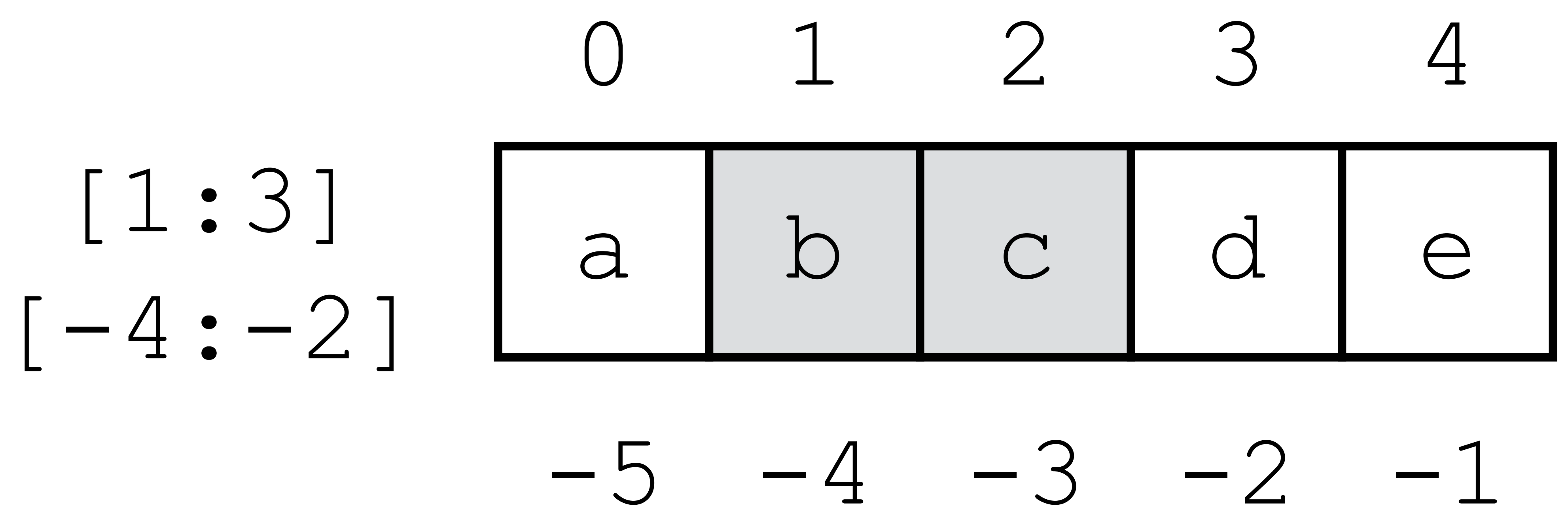
Indexing (Positive and Negative)

- Positive indices start at zero, negative at -1
- `my_str = "abcde"; my_str[1] # "b"`
- `my_list = [1,2,3,4,5]; my_list[-3] # 3`
- `my_tuple = (1,2,3,4,5); my_tuple[-5] # 1`



Slicing

- Positive or negative indices can be used at any step
- `my_str = "abcde"; my_str[1:3] # ["b", c"]`
- `my_list = [1,2,3,4,5]; my_list[3:-1] # [4]`
- Implicit indices
  - `my_tuple = (1,2,3,4,5); my_tuple[-2:] # (4,5)`
  - `my_tuple[:3] # (1,2,3)`



Iteration

-
  ```
  for d in sequence:
      # do stuff
  ```
- **Important**: `d` is a **data** item, **not** an **index**!

-
  ```
  sequence = "abcdef"
  for d in sequence:
      print(d, end=" ")                   # a b c d e f
  ```

-
  ```
  sequence = [1,2,3,4,5]
  for d in sequence:
      print(d, end=" ")                   # 1 2 3 4 5
  ```

-
  ```
  sequence = (1,2,3,4,5)
  for d in sequence:
      print(d, end=" ")                   # 1 2 3 4 5
  ```

Membership

- `<expr> in <seq>`
- Returns `True` if the expression is in the sequence, `False` otherwise
- `"a" in "abcde" # True`
- `0 in [1,2,3,4,5] # False`
- `3 in (3, 3, 3, 3) # True`

Sequence Operations

Operator	**Meaning**
`<seq> + <seq>`	Concatenation
`<seq> * <int-expr>`	Repetition
`<seq>[<int-expr>]`	Indexing
`len(<seq>)`	Length
`<seq>[<int-expr?>:<int-expr?>]`	Slicing
`for <var> in <seq>:`	Iteration
`<expr> in <seq>`	Membership (Boolean)

Sequence Operations

Operator	**Meaning**
`<seq> + <seq>`	Concatenation
`<seq> * <int-expr>`	Repetition
`<seq>[<int-expr>]`	Indexing
`len(<seq>)`	Length
`<seq>[<int-expr?>:<int-expr?>]`	Slicing
`for <var> in <seq>:`	Iteration
`<expr> in <seq>`	Membership (Boolean)

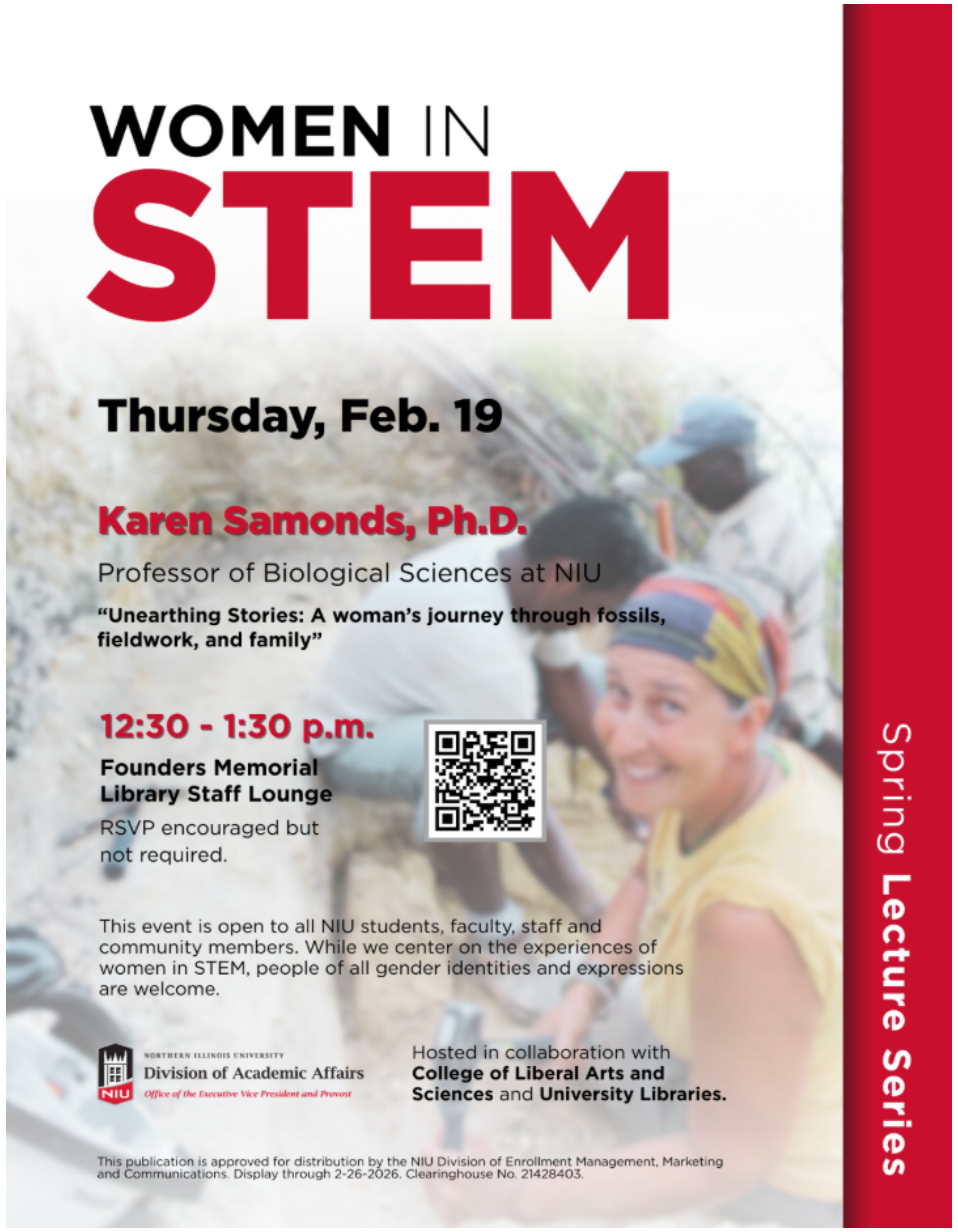
`<int-expr?>`: may be `<int-expr>` but also can be empty

Assignment 2

- Due tonight
- FRACTRAN
- Control Flow and Functions
- Do not use sequences, other collections, or comprehensions for this assignment (except extra credit)

Women in STEM Spring Lecture Series

- Thursday, Feb. 19
- Lecture Series hosted in collaboration with the college and libraries
- All are welcome



WOMEN IN

STEM

Thursday, Feb. 19

Karen Samonds, Ph.D.

Professor of Biological Sciences at NIU

"Unearthing Stories: A woman's journey through fossils, fieldwork, and family"

12:30 - 1:30 p.m.

Founders Memorial Library Staff Lounge

RSVP encouraged but not required.

This event is open to all NIU students, faculty, staff and community members. While we center on the experiences of women in STEM, people of all gender identities and expressions are welcome.

NORTHERN ILLINOIS UNIVERSITY
Division of Academic Affairs
Office of the Executive Vice President and Provost

Hosted in collaboration with **College of Liberal Arts and Sciences** and **University Libraries.**

This publication is approved for distribution by the NIU Division of Enrollment Management, Marketing and Communications. Display through 2-26-2026. Clearinghouse No. 21428403.

Spring **Lecture Series**

What's the difference between the sequences?

- Strings can only store characters, lists & tuples can store arbitrary values
- Mutability: strings and tuples are **immutable**, lists are **mutable**

-
  ```
  my_list = [1, 2, 3, 4]
  my_list[2] = 300
  my_list # [1, 2, 300, 4]
  ```
- `my_tuple = (1, 2, 3, 4); my_tuple[2] = 300 # TypeError`
- `my_str = "abcdef"; my_str[0] = "z" # TypeError`

List methods

Method	**Meaning**
`<list>.append(d)`	Add element `d` to end of list.
`<list>.extend(s)`	Add **all** elements in `s` to end of list.
`<list>.insert(i, d)`	Insert `d` into list at index i.
`<list>.pop(i)`	Deletes `i`th element of the list and returns its value.
`<list>.sort()`	Sort the list.
`<list>.reverse()`	Reverse the list.
`<list>.remove(d)`	Deletes first occurrence of `d` in list.
`<list>.index(d)`	Returns index of first occurrence of `d`.
`<list>.count(d)`	Returns the number of occurrences of `d` in list.

List methods

Method	Meaning Mutate
`<list>.append(d)`	Add element `d` to end of list.
`<list>.extend(s)`	Add **all** elements in `s` to end of list.
`<list>.insert(i, d)`	Insert `d` into list at index i.
`<list>.pop(i)`	Deletes `i`th element of the list and returns its value.
`<list>.sort()`	Sort the list.
`<list>.reverse()`	Reverse the list.
`<list>.remove(d)`	Deletes first occurrence of `d` in list.
`<list>.index(d)`	Returns index of first occurrence of `d`.
`<list>.count(d)`	Returns the number of occurrences of `d` in list.

The del statement

- `pop` works well for removing an element by index plus it **returns** the element
- Can also remove an element at index `i` using
  - `del my_list[i]`
- Note this is very different syntax so I prefer `pop`
- But del can **delete slices**
  - `del my_list[i:j]`
- Also, can delete **identifier** names completely

  -
    ```
    a = 32
    del a
    a # NameError
    ```
- This is different than `a = None`

Updating collections

- There are three ways to deal with operations that update collections:
  - Returns an updated **copy** of the list
  - Updates the collection **in place**
  - Updates the collection in place **and returns it**
- `list.sort` and `list.reverse` work **in place** and **don't return** the list
- Common error:
  - `sorted_list = my_list.sort() # sorted_list = None`
- Instead:
  - `sorted_list = sorted(my_list)`

sorted and reversed

- For both sort and reverse, have `sorted` & `reversed` which are **not** in place
- Called with the sequence as the argument

-
  ```
  my_list = [7, 3, 2, 5, 1]
  for d in sorted(my_list):
      print(d, end=" ")                # 1 2 3 5 7
  ```

-
  ```
  my_list = [7, 3, 2, 5, 1]
  for d in reversed(my_list):
      print(d, end=" ")                # 1 5 2 3 7
  ```
- But this doesn't work:
  - `reversed_list = reversed(my_list)`
- If you need a new list (same as with `range`):
  - `reversed_list = list(reversed(my_list))`

Reversed sort

- Both sort and sorted have a boolean parameter `reverse` that will sort the list in reverse

-
  ```
  my_list = [7, 3, 2, 5, 1]
  my_list.sort(reverse=True) # my_list now [7, 5, 3, 2, 1]
  ```

-
  ```
  for i in sorted(my_list, reverse=True):
      print(i, end = " ")    # prints 7 5 3 2 1
  ```
- There is also a `key` parameter that should be a **function** that will be called on each element before comparisons—the outputs will be used to sort
  - Example: convert to lowercase

Nested Sort

- By default, sorts by comparing inner elements in order
- `sorted([[4,2],[1,5],[1,3],[3,5]])`
  - 1st element: `1 == 1 < 3 < 4`
  - 2nd element for equal: `3 < 5`
  - Result: `[[1,3],[1,5],[3,5],[4,2]]`
- Longer lists after shorter lists:
  - `sorted([[1,2],[1]]) # [[1],[1,2]]`

enumerate

- Often you **do not** need the index when iterating through a sequence
- If you need an index while looping through a sequence, use `enumerate`

-
  ```
  for i, d in enumerate(my_list):
      print("index:", i, "element:", d)
  ```
- Each time through the loop, it yields **two** items, the **index** `i` & the **element** `d`
- `i, d` is actually a **tuple**
- Automatically **unpacked** above, can manually do this, but don't!

-
  ```
  for t in enumerate(my_list):
      i = t[0]
      d = t[1]
      print("index:", i, "element:", d)
  ```

enumerate

- Often you **do not** need the index when iterating through a sequence
- If you need an index while looping through a sequence, use `enumerate`

-
  ```
  for i, d in enumerate(my_list):
      print("index:", i, "element:", d)
  ```
- Each time through the loop, it yields **two** items, the **index** `i` & the **element** `d`
- `i, d` is actually a **tuple**
- Automatically **unpacked** above, can manually do this, but don't!

-
  ```
  for t in enumerate(my_list):
      i = t[0]
      d = t[1]
      print("index:", i, "element:", d)
  ```

Tuples

- Tuples are **immutable** sequences
- We've actually seen tuples a couple of times already
  - Simultaneous Assignment
  - Returning Multiple Values from a Function
- Python allows us to omit parentheses when it's clear

```
- b, a = a, b         # same as (b, a) = (a, b)
- t1 = a, b           # don't normally do this
- c, d = f(2, 5, 8) # same as (c, d) = f(2, 5, 8)
- t2 = f(2, 5, 8)   # don't normally do this
```

Tuple Packing and Unpacking

-
  ```
  def f(a, b):
      if a > 3:
          return a, b-a # tuple packing
      return a+b, b # tuple packing
  ```
- `c, d = f(4, 3) # tuple unpacking`

- Make sure to unpack the correct number of variables!
- `c, d = a+b, a-b, 2*a # ValueError: too many values to unpack`
- Sometimes, check return value before unpacking:

  -
    ```
    retval = f(42)
    if retval is not None:
        c, d = retval
    ```

Tuple Packing and Unpacking

-
  ```
  def f(a, b):
      if a > 3:
          return a, b-a # tuple packing
      return a+b, b # tuple packing
  ```

```
t = (a, b-a)
  return t
```

- `c, d = f(4, 3) # tuple unpacking`

- Make sure to unpack the correct number of variables!
- `c, d = a+b, a-b, 2*a # ValueError: too many values to unpack`
- Sometimes, check return value before unpacking:

  -
    ```
    retval = f(42)
    if retval is not None:
        c, d = retval
    ```

Tuple Packing and Unpacking

-
  ```
  def f(a, b):
      if a > 3:
          return a, b-a # tuple packing
      return a+b, b # tuple packing
  ```

```
t = (a, b-a)
return t
```

- `c, d = f(4, 3) # tuple unpacking`

```
t = f(4, 3)
(c, d) = t
```

- Make sure to unpack the correct number of variables!
- `c, d = a+b, a-b, 2*a # ValueError: too many values to unpack`
- Sometimes, check return value before unpacking:

  -
    ```
    retval = f(42)
    if retval is not None:
        c, d = retval
    ```

Unpacking other sequences

- You can unpack other sequences, too
  - `a, b = 'ab'`
  - `a, b = ['a', 'b']`
- Why is list unpacking rare?

Unpacking other sequences

- You can unpack other sequences, too
  - `a, b = 'ab'`
  - `a, b = ['a', 'b']`
- Why is list unpacking rare?
  - Lists are mutable, assignment is generally static
  - But can use a star to capture a variable number of values
    - `a, *b, c = ['a', 'b', …, 'y', 'z'] # b is ['b', …, 'y']`

Other sequence methods

- `my_list = [7, 2, 1, 12]`
- Math methods:
  - `max(my_list) # 12`
  - `min(my_list) # 1`
  - `sum(my_list) # 22`
- `zip`: combine two sequences into a single sequence of tuples

  -
    ```
    zip_list = list(zip(my_list, "abcd"))
    zip_list # [(7, 'a'), (2, 'b'), (1, 'c'), (12, 'd')]
    ```
  - Use this instead of using indices to count through both

Functions

Functions

- Call a function `f`: `f(3)` or `f(3,4)` or … depending on number of parameters

-
  ```
  def <function-name>(<parameter-names>):
      """Optional docstring documenting the function"""
      <function-body>
  ```
- `def` stands for function definition
- docstring is convention used for documentation
- Remember the **colon** and **indentation**
- Parameter list can be empty: `def f(): …`

Functions

- Use `return` to return a value

-
  ```
  def <function-name>(<parameter-names>):
      # do stuff
      return res
  ```
- Can return more than one value using commas

-
  ```
  def <function-name>(<parameter-names>):
      # do stuff
      return res1, res2
  ```
- Use **simultaneous assignment** when calling:
  - `a, b = do_something(1,2,5)`
- If there is no return value, the function returns `None` (a special value)

Return

- As many return statements as you want
- Always end the function and go back to the calling code
- Returns do not need to match one type/structure (generally not a good idea)

-
  ```
  def f(a,b):
      if a < 0:
          return -1
      while b > 10:
          b -= a
          if b < 0:
              return "BAD"
      return b
  ```

Scope

- The **scope** of a variable refers to where in a program it can be referenced
- Python has three scopes:
  - **global**: defined outside a function
  - **local**: in a function, only valid in the function
  - **nonlocal**: can be used with nested functions
- Python allows variables in different scopes to have the **same name**

Global read

-
  ```
  def f(): # no arguments
      print("x in function:", x)

  x = 1
  f()
  print("x in main:", x)
  ```
- Output:

  -
    ```
    x in function: 1
    x in main: 1
    ```
- Here, the `x` in `f` is read from the global scope

Try to modify global?

-
  ```
  def f(): # no arguments
      x = 2
      print("x in function:", x)

  x = 1
  f()
  print("x in main:", x)
  ```
- Output:

  -
    ```
    x in function: 2
    x in main: 1
    ```
- Here, the `x` in `f` is in the local scope

Global keyword

-
  ```
  def f(): # no arguments
      global x
      x = 2
      print("x in function:", x)

  x = 1
  f()
  print("x in main:", x)
  ```
- Output:

  -
    ```
    x in function: 2
    x in main: 2
    ```
- Here, the `x` in `f` is in the global scope because of the global declaration

What is the scope of a parameter of a function?