Programming Principles in Python (CSCI 503)

Sequences

Dr. David Koop

(some slides adapted from Dr. Reva Freedman)

if, else, elif, pass

- ```
  if a < 10:
      print("Small")
  else:
      if a < 100:
          print("Medium")
      else:
          if a < 1000:
              print("Large")
          else:
              print("X-Large")
  ```
- ```
  if a < 10:
      print("Small")
  elif a < 100:
      print("Medium")
  elif a < 1000:
      print("Large")
  else:
      print("X-Large")
  ```

- Indentation is critical so else-if branches can become unwieldy (elif helps)
- Remember colons and indentation
- `pass` can be used for an empty block

while, break, continue

- `while <boolean expression>:`
  `    <loop-block>`

```
while <boolean expression>:
    <loop-block>
```

- Condition is checked at the **beginning** and before each repeat
- `break`: **immediately** exit the current loop
- `continue`: stop loop execution and go back to the **top** of the loop, checking the condition again
- while d > 0:

```
while d > 0:
    a = get_next_input()
    if a > 100:
        break
    if a < 10:
        continue
    d -= a
```

The Go To Statement Debate

"…I became convinced that the go to statement should be abolished from all 'higher level' programming languages… The go to statement as it stands is just too primitive; it is too much an invitation to make a mess of one's program."

Go To Statement Considered Harmful

Key Words and Phrases: go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing

CR Categories: 4.22, 5.23, 5.24

EDITOR:

For a number of years I have been familiar with the observation

been urged to do so.

My first remark is that, although the programmer's activity ends when he has constructed a correct program, the process taking place under control of his program is the true subject matter of his activity, for it is this process that has to accomplish the desired effect; it is this process that in its dynamic behavior has to satisfy the desired specifications. Yet, once the program has been made, the "making" of the corresponding process is delegated to the machine.

My second remark is that our intellectual powers are rather

dynamic progress is only characterized when we also give to which call of the procedure we refer. With the inclusion of procedures we can characterize the progress of the process via a sequence of textual indices, the length of this sequence being equal to the dynamic depth of procedure calling.

Let us now consider repetition clauses (like, **while** *B* **repeat** *A* or **repeat** *A* **until** *B*). Logically speaking, such clauses are now superfluous, because we can express repetition with the aid of

namic index," inexorably counting the ordinal number of the corresponding current repetition. As repetition clauses (just as procedure calls) may be applied nestedly, we find that now the progress of the process can always be uniquely characterized by a (mixed) sequence of textual and/or dynamic indices.

The main point is that the values of these indices are outside programmer's control; they are generated (either by the write-up of his program or by the dynamic evolution of the process) whether he wishes or not. They provide independent coordinates in which to describe the progress of the process.

[Dijkstra, 1968]

Loop Styles

- Loop-and-a-Half

```
d = get_data() # priming rd
while check(d):
    # do stuff
    d = get_data()
```

- Infinite-Loop-Break

```
while True:
    d = get_data()
    if check(d):
        break
    # do stuff
```

- Assignment Expression (Walrus)

```
while check(d := get_data):
    # do stuff
```

For Loop

- for loops in Python are really for-each loops
- Always an element that is the current element
  - Can be used to iterate through iterables (containers, generators, strings)
  - Can be used for counting
- ```
  for i in range(5):
      print(i) # 0 1 2 3 4
  ```
- `range` generates the sequences of integers, one at a time
  - `range(n)` ➞ `0, 1, …, n-1`
  - `range(start, n)` ➞ `start, start + 1, …, start + (n-1)`
  - `range(start, n, step)`
    ➞ `start, start + step, …, start + (n-1)*step`
  ```

Assignment 2

- Due Monday
- Python control flow and functions
- Do not use containers like lists (except for the extra credit)!
- Simple FRACTRAN programs
- Make sure to follow instructions
  - Name the submitted file a2.ipynb
  - Put your name and z-id in the first cell
  - Label each part of the assignment using markdown
  - Make sure to produce output according to specifications

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)

Default Values & Keyword Arguments

- Can add `=<value>` to parameters
- ```
  def rectangle_area(width=30, height=20):
      return width * height
  ```
- All of these work:
  - `rectangle_area() # 600`
  - `rectangle_area(10) # 200`
  - `rectangle_area(10,50) # 500`
- If the user does not pass an argument for that parameter, the parameter is set to the default value
- Can also pass parameters using `<name>=<value>` (keyword arguments):
  - `rectangle_area(height=50) # 1500`
  ```

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
```

Sequences

- Strings are sequences of characters: "abcde"
- Lists are also sequences: [1, 2, 3, 4, 5]
- + Tuples: (1, 2, 3, 4, 5)

Lists

- 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]]`

~~Lists~~ Tuples

- Defining a tuple: `my_tuple = (0, 1, 2, 3, 4)`
- But tuples can store different types:
  - `my_tuple = (0, "a", 1.34)`
- Including other tuples:
  - `my_tuple = (0, "a", 1.34, (1, 2, 3))`
- How do you define a tuple with **one** element?

~~Lists~~ Tuples

- Defining a tuple: `my_tuple = (0, 1, 2, 3, 4)`
- But tuples can store different types:
  - `my_tuple = (0, "a", 1.34)`
- Including other tuples:
  - `my_tuple = (0, "a", 1.34, (1, 2, 3))`
- How do you define a tuple with **one** element?
  - `my_tuple = (1)  # doesn't work`
  - `my_tuple = (1,) # add trailing comma`

List Operations

- **Not** like vectors or matrices!
- 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`

~~List~~ 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`

Sequence Indexing

- Square brackets are used to pull out an element of a sequence
- We always start counting at **zero**!
- `my_str = "abcde"; my_str[0] # "a"`
- `my_list = [1,2,3,4,5]; my_list[2] # 3`
- `my_tuple = (1,2,3,4,5); my_tuple[5] # IndexError`

0	1	2	3	4
a	b	c	d	e

Negative Indexing

- Subtract from the end of the sequence to the beginning
- We always start counting at ~~zero~~ -1 (zero would be ambiguous!)
- `my_str = "abcde"; my_str[-1] # "e"`
- `my_list = [1,2,3,4,5]; my_list[-3] # 3`
- `my_tuple = (1,2,3,4,5); my_tuple[-5] # 1`

0	1	2	3	4
a	b	c	d	e
-5	-4	-3	-2	-1

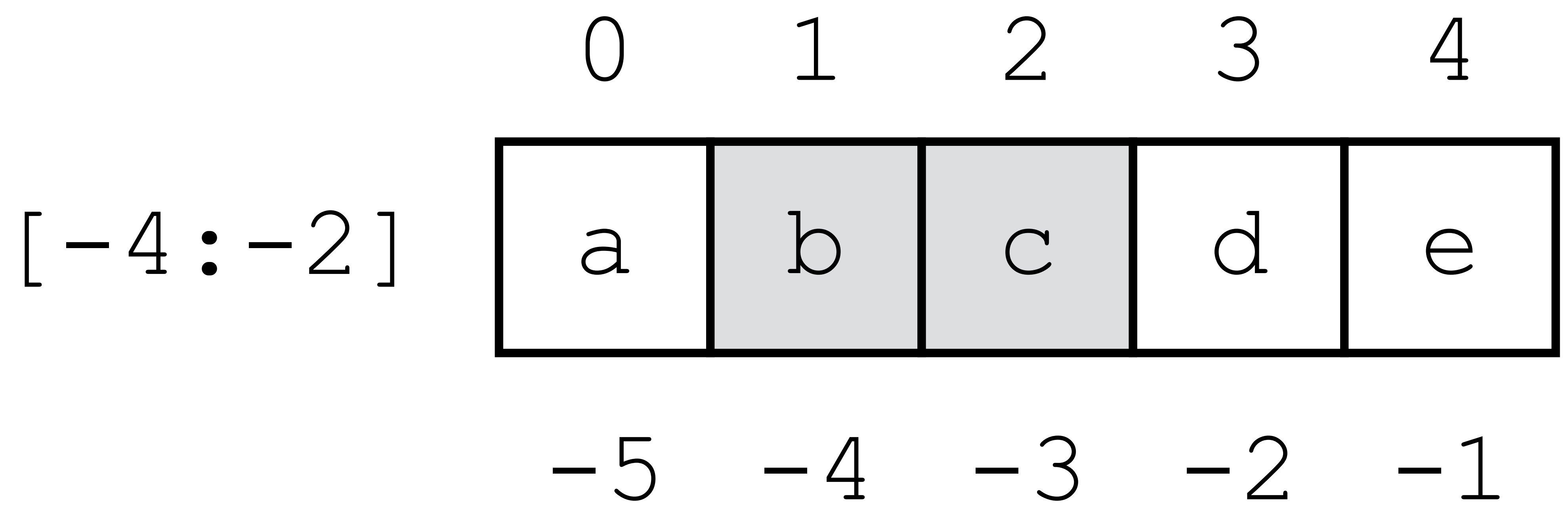
Slicing

- Want a subsequence of the given sequence
- Specify the start and the first index not included
- Returns the same type of sequence
- `my_str = "abcde"; my_str[1:3] # "bc"`
- `my_list = [1,2,3,4,5]; my_list[3:4] # [4]`
- `my_tuple = (1,2,3,4,5); my_tuple[2:99] # (3,4,5)`

	0	1	2	3	4
[1:3]	a	b	c	d	e

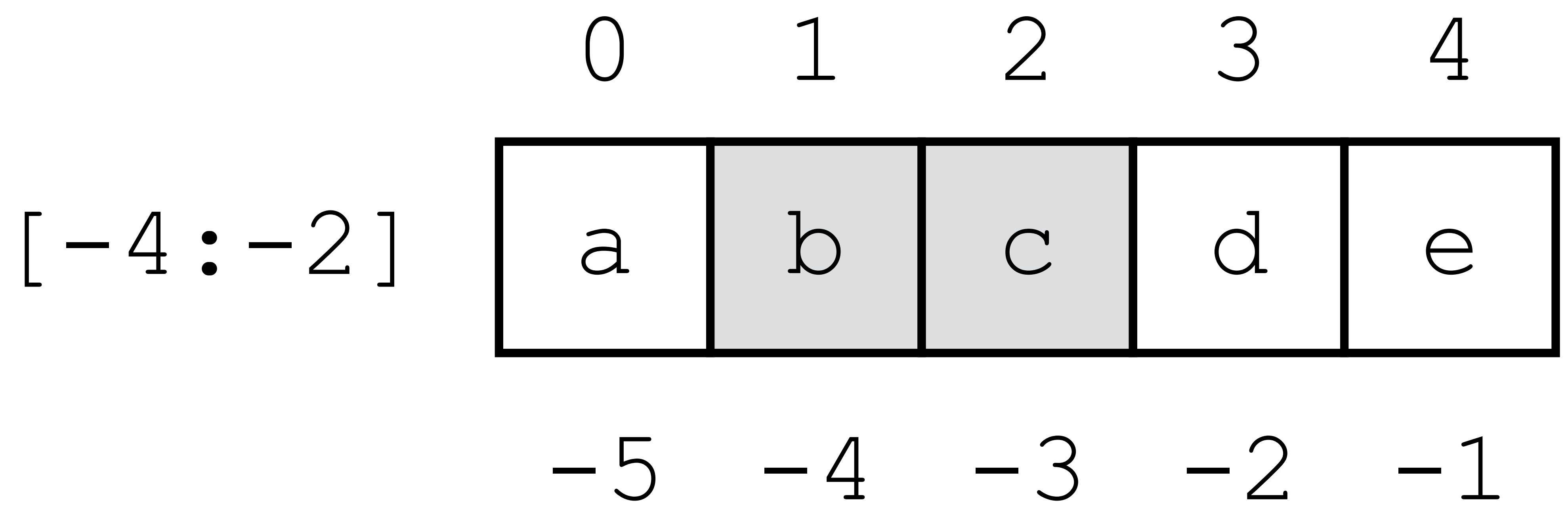
Negative Indices with Slices

- Negative indices can be used instead or with non-negative indices
- `my_str = "abcde"; my_str[-4:-2] # "bc"`
- `my_list = [1,2,3,4,5]; my_list[3:-1] # [4]`
- How do we include the last element?
- `my_tuple = (1,2,3,4,5); my_tuple[-2:?]`



Negative Indices with Slices

- Negative indices can be used instead or with non-negative indices
- `my_str = "abcde"; my_str[-4:-2] # "bc"`
- `my_list = [1,2,3,4,5]; my_list[3:-1] # [4]`
- How do we include the last element?
- `my_tuple = (1,2,3,4,5); my_tuple[-2:?]`

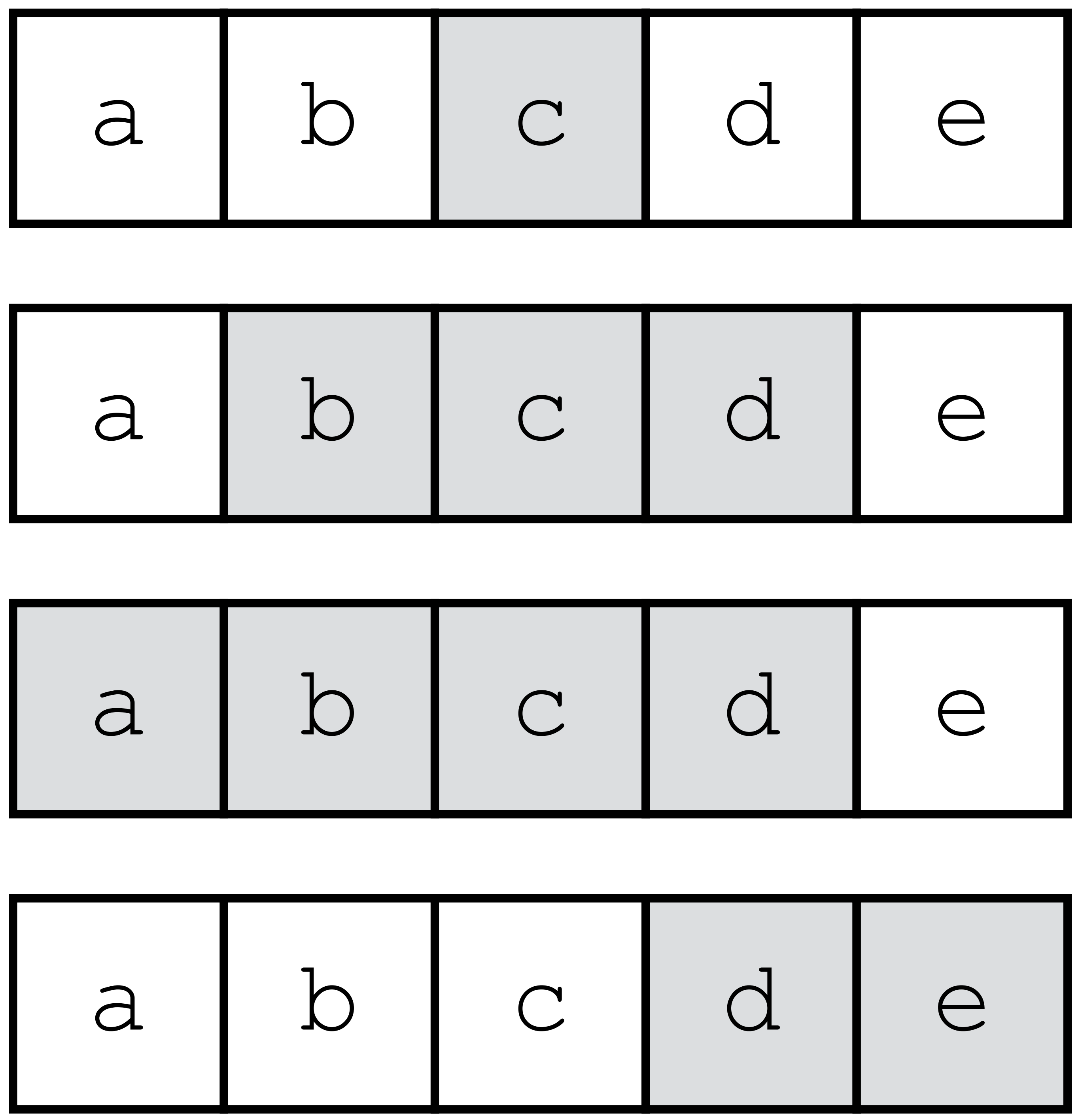


Implicit Indices

- Don't need to write indices for the beginning or end of a sequence
- Omitting the first number of a slice means start from the beginning
- Omitting the last number of a slice means go through the end
- `my_tuple = (1,2,3,4,5);` ~~`my_tuple[-2:len(my_tuple)]`~~
- `my_tuple = (1,2,3,4,5); my_tuple[-2:] # (4,5)`
- Can create a **copy** of a sequence by omitting both
- `my_list = [1,2,3,4,5]; my_list[:] # [1,2,3,4,5]`

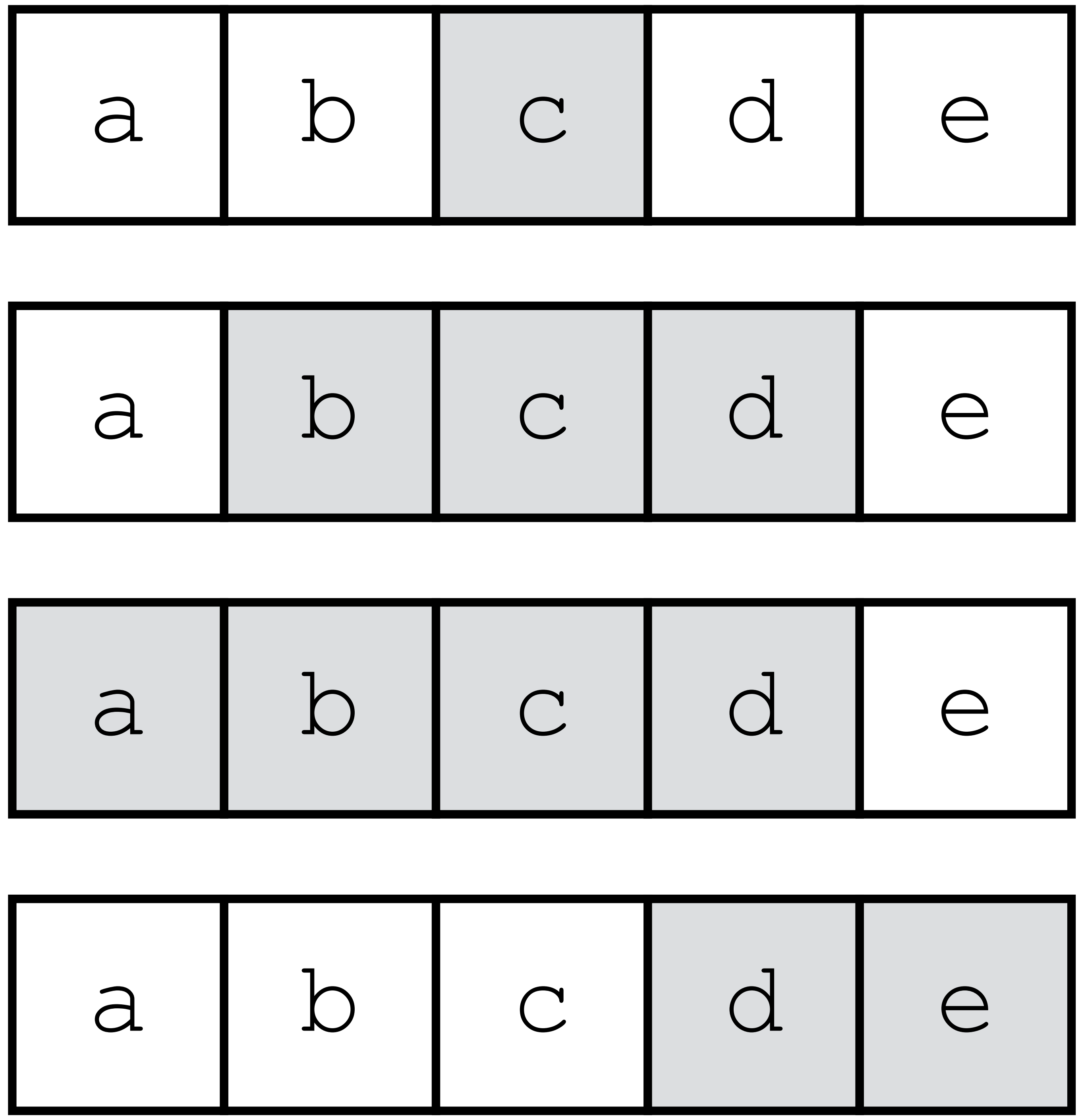
Indexing Quiz

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



Indexing Quiz

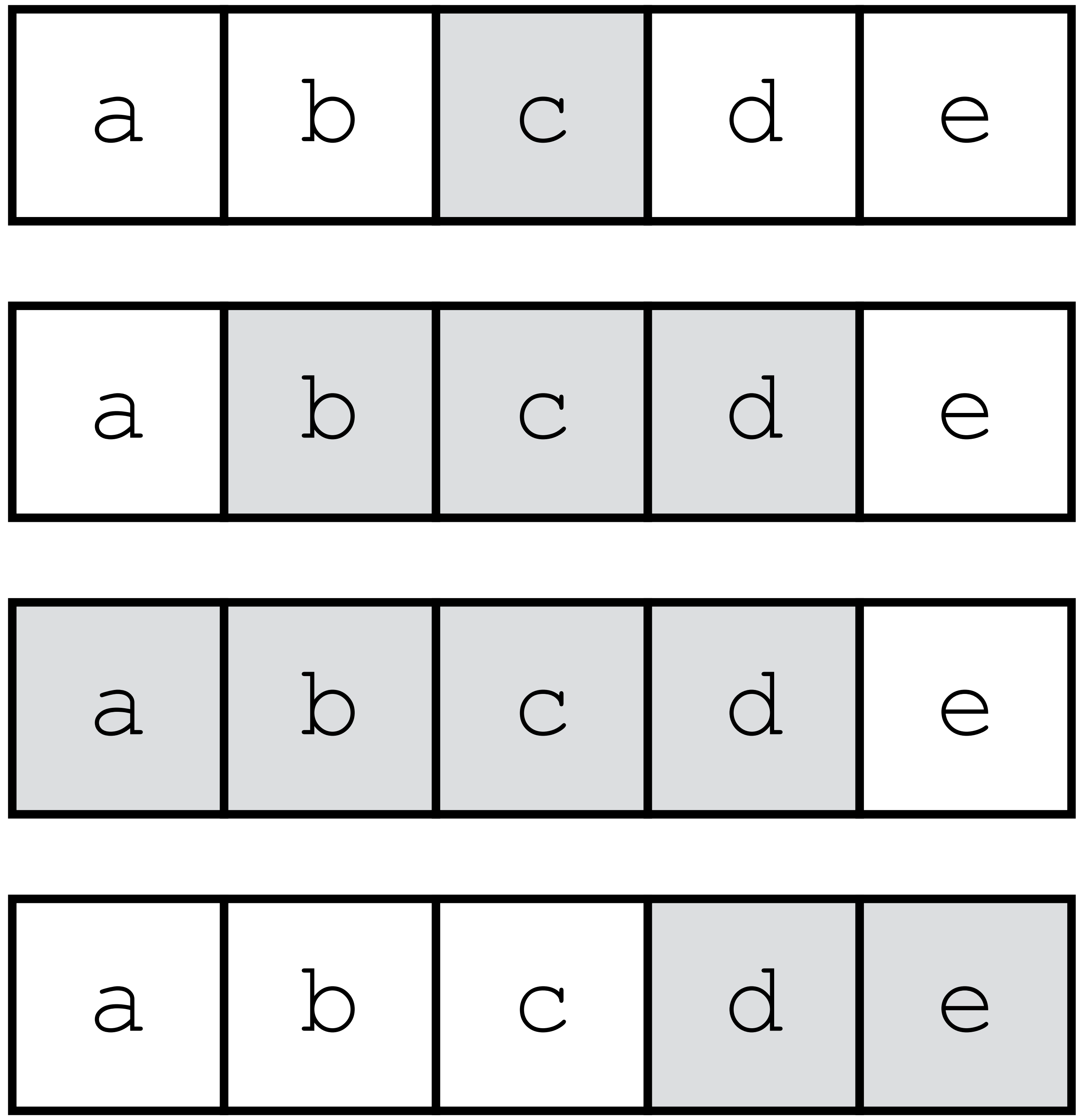
```
my_list = ['a', 'b', 'c', 'd', 'e']
```




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

Indexing Quiz

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

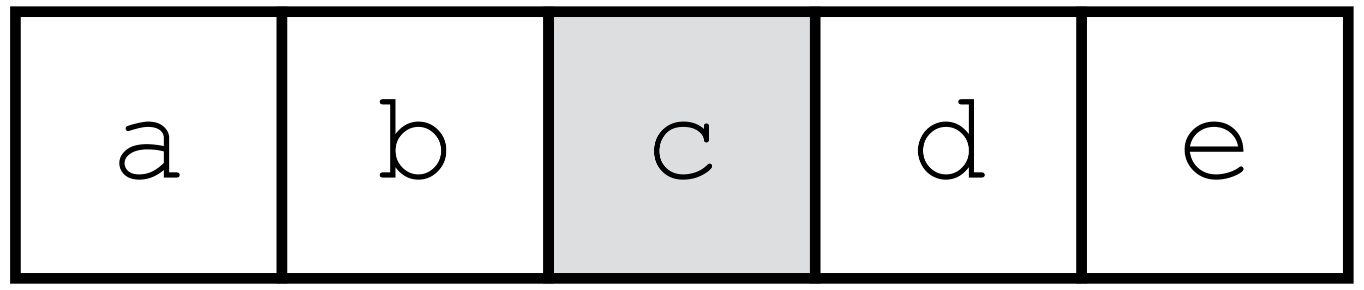



```
my_list[2]; my_list[-3]; my_list[2:3]
```

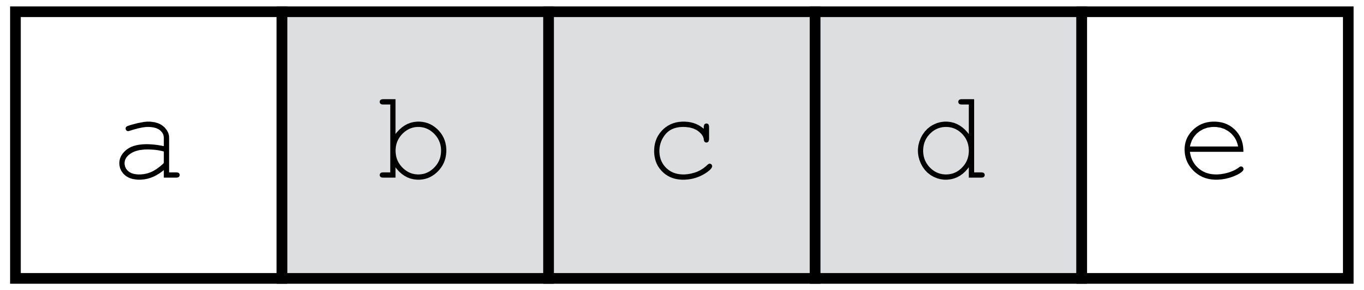
```
my_list[1:4]; my_list[-4:-1];
my_list[1:-1]
```

Indexing Quiz

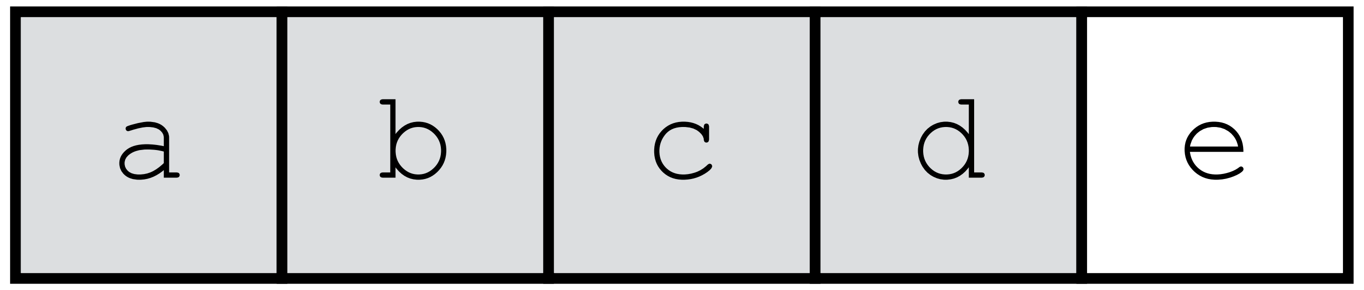
```
my_list = ['a', 'b', 'c', 'd', 'e']
```



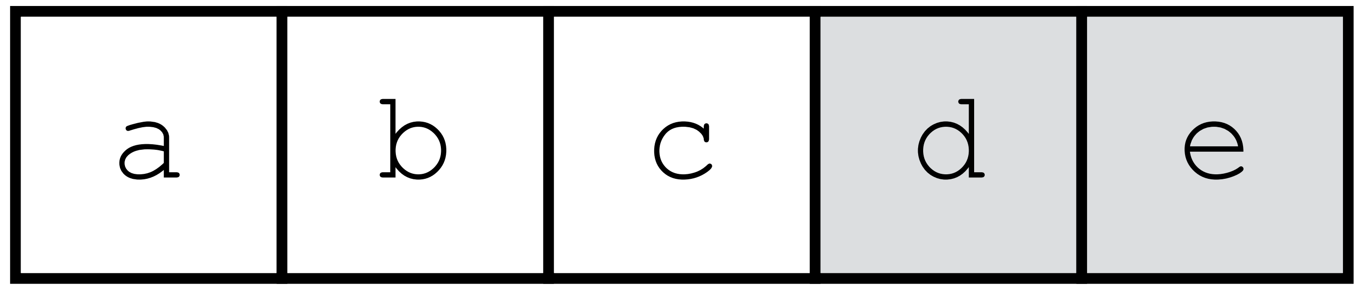

```
my_list[2]; my_list[-3]; my_list[2:3]
```




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

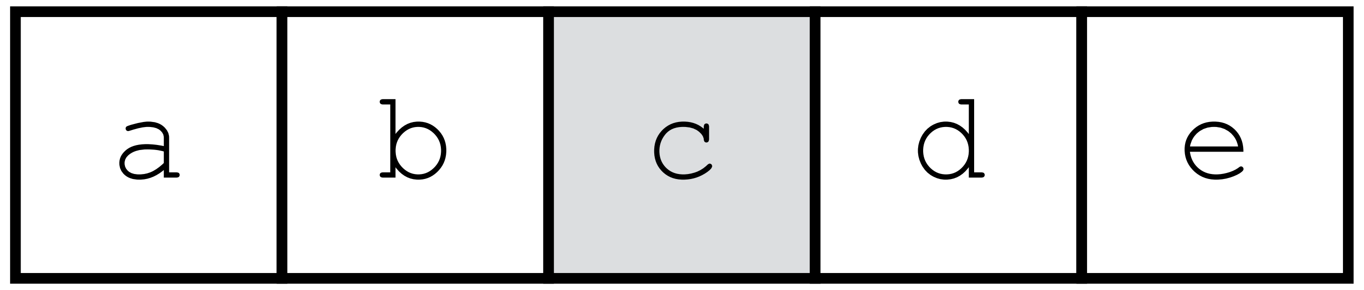



```
my_list[0:4]; my_list[:4];
my_list[-5:-1]
```

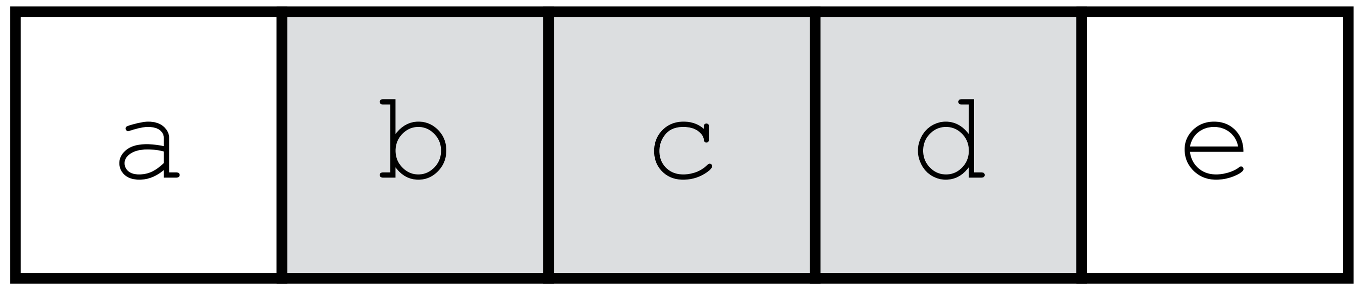


Indexing Quiz

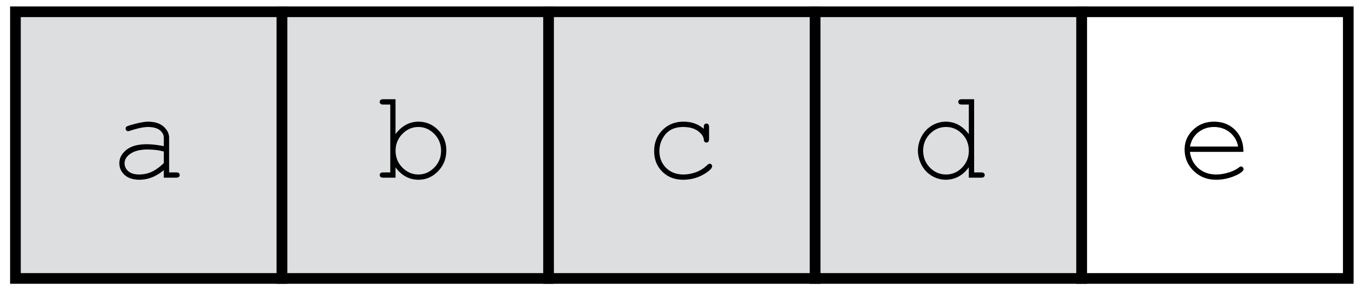
```
my_list = ['a', 'b', 'c', 'd', 'e']
```



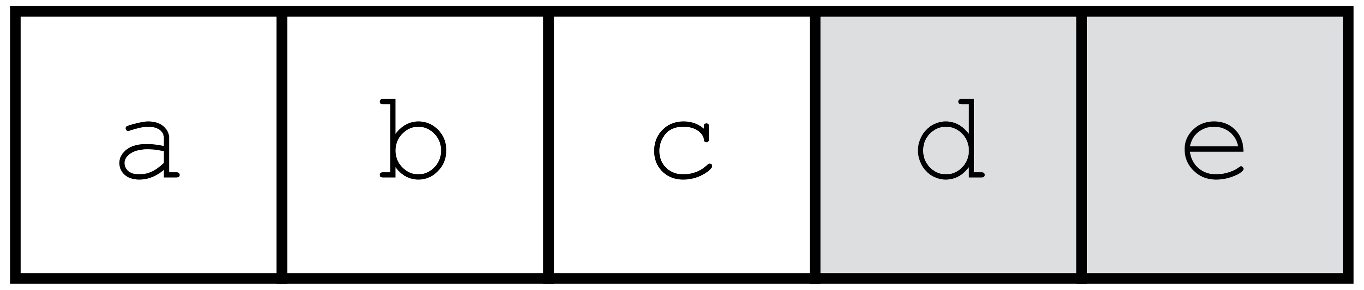

```
my_list[2]; my_list[-3]; my_list[2:3]
```




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




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




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

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

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 ith 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.