

Programming Principles in Python (CSCI 503/490)

Sequences

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']
```

a	b	c	d	e
---	---	---	---	---

a	b	c	d	e
---	---	---	---	---

a	b	c	d	e
---	---	---	---	---

a	b	c	d	e
---	---	---	---	---

Indexing & Slicing Quiz

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

a	b	c	d	e
---	---	---	---	---

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

a	b	c	d	e
---	---	---	---	---

a	b	c	d	e
---	---	---	---	---

a	b	c	d	e
---	---	---	---	---

Indexing & Slicing Quiz

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

a	b	c	d	e
---	---	---	---	---

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

a	b	c	d	e
---	---	---	---	---

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

a	b	c	d	e
---	---	---	---	---

a	b	c	d	e
---	---	---	---	---

Indexing & Slicing Quiz

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

a	b	c	d	e
---	---	---	---	---

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

a	b	c	d	e
---	---	---	---	---

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

a	b	c	d	e
---	---	---	---	---

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

a	b	c	d	e
---	---	---	---	---

Indexing & Slicing Quiz

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

a	b	c	d	e
---	---	---	---	---

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

a	b	c	d	e
---	---	---	---	---

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

a	b	c	d	e
---	---	---	---	---

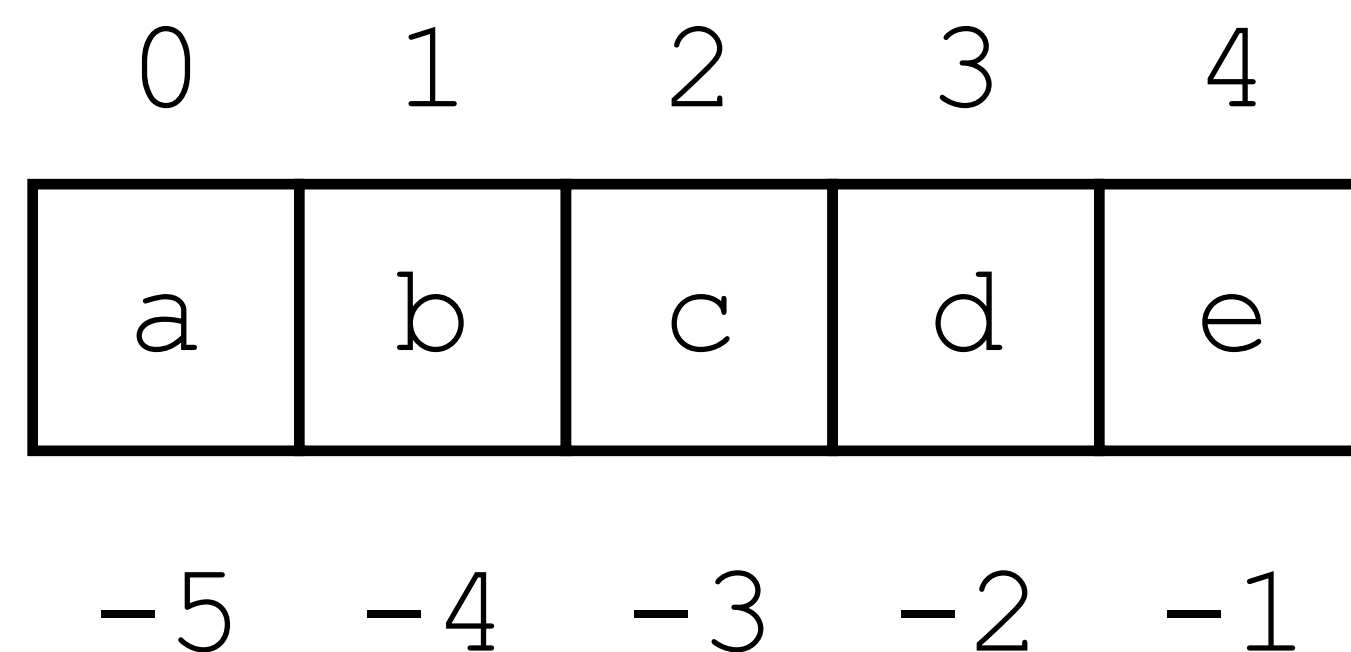
```
my_list[0:4]; my_list[:4];  
my_list[-5:-1]
```

a	b	c	d	e
---	---	---	---	---

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

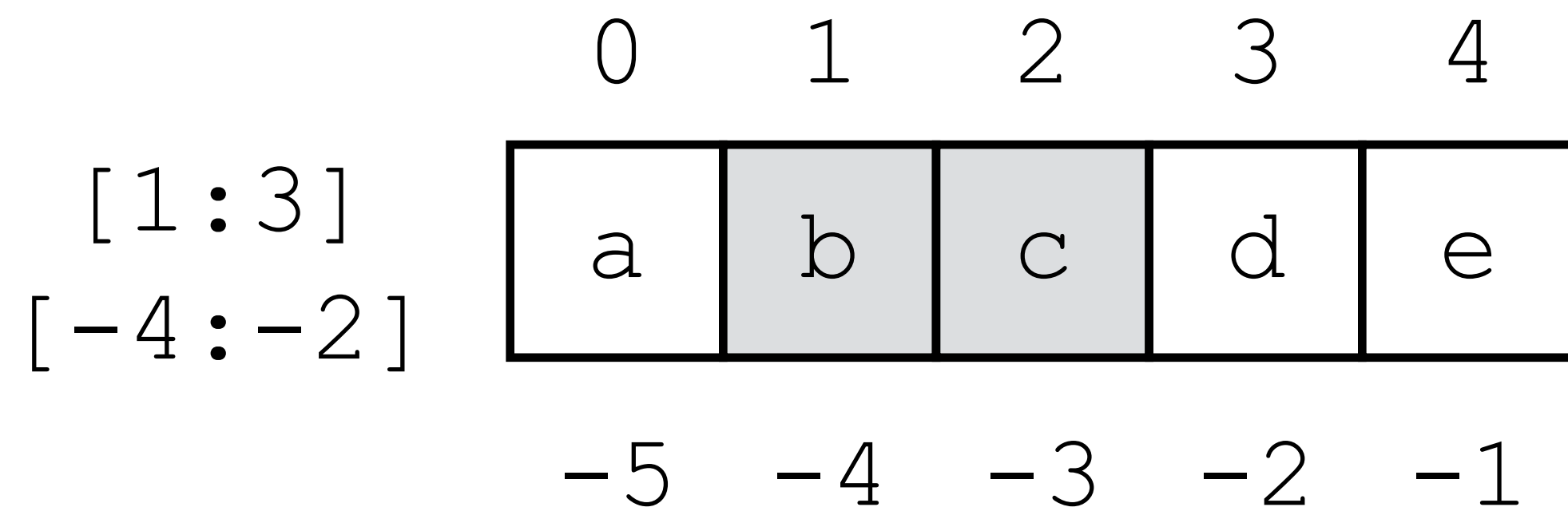
Indexing (Positive and Negative)

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

Sequence Operations

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

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

Assignment 2

- Due tomorrow
- Grades and feedback from Assignment 1 are posted
- Python control flow and functions
- **Do not** use sequences or comprehensions!
- 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

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                                                              |
|----------------------------------------|----------------------------------------------------------------------|
| <code>&lt;list&gt;.append(d)</code>    | Add element <code>d</code> to end of list.                           |
| <code>&lt;list&gt;.extend(s)</code>    | Add <b>all</b> elements in <code>s</code> to end of list.            |
| <code>&lt;list&gt;.insert(i, d)</code> | Insert <code>d</code> into list at index <code>i</code> .            |
| <code>&lt;list&gt;.pop(i)</code>       | Deletes <code>i</code> th element of the list and returns its value. |
| <code>&lt;list&gt;.sort()</code>       | Sort the list.                                                       |
| <code>&lt;list&gt;.reverse()</code>    | Reverse the list.                                                    |
| <code>&lt;list&gt;.remove(d)</code>    | Deletes first occurrence of <code>d</code> in list.                  |
| <code>&lt;list&gt;.index(d)</code>     | Returns index of first occurrence of <code>d</code> .                |
| <code>&lt;list&gt;.count(d)</code>     | Returns the number of occurrences of <code>d</code> in list.         |

# List methods

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

# 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
```

# 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
```

# 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?

# 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