CS134 Lecture 15:
Tuples and Set Examples
Announcements & Logistics

• No HW due next Monday

• **Midterm reminders:**
  
  • **Review:** *Monday 3/11* from 7-9pm
  
  • **Exam** *Thurs 3/14* from 6-7:30pm OR 8-9:30pm
  
  • Both exam and review are in Bronfman Auditorium
  
  • Exam only includes material up to this week
  
  • *Sample Exam* posted!

• New Instructor Help Hours Schedule
  
  • Wednesday 1-4, Thursday 1-4

---

**Do You Have Any Questions?**
Last Time: Aliasing

- Describe how scope works when lists are passed as function parameters (interaction between scope and aliasing)
- Explore two new Python types:
  - tuples: *immutable ordered* alternative to lists
  - sets: *mutable unordered* collection (if time permits)
Today's Plan

• Finish up presentation of sets
• Write some code together (using tuples and sets) to solve familiar problems.
Sets
New Unordered Data Structure: Sets

- Sets are *mutable*, *unordered* collections of *immutable* objects
  - Sets can change (e.g., we can add and remove items), but an item cannot be changed once the item is added to the set
- Sets are written as comma separated values between curly braces \{ \}
- Elements in a set must be *unique* and *immutable*
  - Sets can be an effective way of **eliminating duplicate values**

```python
>>> nums = {42, 17, 8, 57, 23}
>>> flowers = {"tulips", "daffodils", "asters", "daisies"}
>>> empty_set = set() # empty set
```
New Unordered Data Structure: Sets

• **Question:** What is the potential downside of removing duplicates w/sets?

```python
>>> first_choice = {'a', 'b', 'a', 'a', 'b', 'c'}
>>> uniques = set(first_choice)
>>> uniques
# ???
>>> set("aabrackadabra")
# ???
```
New Unordered Data Structure: Sets

**Question:** What is the potential downside of removing duplicates w/sets?

- Might lose the ordering of elements

```python
>>> first_choice = {'a', 'b', 'a', 'a', 'b', 'c'}
>>> uniques = set(first_choice)
>>> uniques
{'a', 'b', 'c'}
>>> set("aabrackadabra")
{'a', 'b', 'd', 'k', 'r'}
```
Sets: Creating New Sets

- There are two ways to create a new set:
  - By placing curly brackets around elements:
    ```
    >>> set_brack = {'aardvark'}
    >>> set_brack
    {'aardvark'}
    ```
  - By converting an iterable collection into a set:
    ```
    >>> set_func = set('aardvark')
    >>> set_func
    {'d', 'v', 'a', 'r', 'k'}
    ```
  - And only one way to create an empty set:
    ```
    >>> empty_set = set()
    >>> empty_set
    set()
    ```

Why letters here instead of the word?

Strings are iterable collection!
### Sets: Membership and Iteration

- Can check membership in a `set` using `in`, `not in`
- Can check length of a set using `len()`
- Can iterate over values in a loop (order will be arbitrary)

```python
>>> nums = {42, 17, 8, 57, 23}
>>> flowers = {"tulips", "daffodils", "asters", "daisies"}
>>> 16 in nums
False
>>> "asters" in flowers
True
>>> len(flowers)
4
>>> # iterable
>>> for f in flowers:
...   print(f)
tulips
daisies
daffodils
asters
```
Sets are Unordered

• Therefore we **cannot**:
  • Index into a set (no notion of “position”)
  • Concatenate (+) two sets (concatenation implies ordering)
  • Create a set of *mutable* objects:
    • Such as lists, sets, and *dictionaries* (foreshadowing...)

```python
>>> {[3, 2], [1, 5, 4]}
TypeError
```

```python
>>> 1 {[3, 2], [1, 5, 4]}
TypeError: unhashable type: 'list'
```
The usual operations you think of in set theory are implemented as follows.

The following operations always return a new set.

- **s1 | s2** (Set Union)
  - Returns a new set that has all elements that are either in **s1** or **s2**

- **s1 & s2** (Set Intersection)
  - Returns a new set that has all the elements that are common to both sets.

- **s1 - s2** (Set Difference)
  - Returns a new set that has all the elements of **s1** that are not in **s2**

**s1 |= s2, s1 &= s2, s1 -= s2** are versions of |, &, - that mutate **s1** to become the result of the operation on the two sets.
Set Operations

```python
>>> cs134_dogs = {"wally", "pixel", "linus", "chelsea", "sally", "artie"}
```

```python
>>> peanuts = {"sally", "linus", "charlie", "franklin", "lucy", "patty"}
```
Set Operations

```python
>>> cs134_dogs = {"wally", "pixel", "linus", "chelsea", "sally", "artie"}
>>> peanuts = {"sally", "linus", "charlie", "franklin", "lucy", "patty"}

>>> union = cs134_dogs | peanuts
>>> union
{'sally', 'wally', 'patty', 'chelsea', 'pixel', 'franklin', 'lucy', 'artie', 'linus', 'charlie'}

>>> intersect = cs134_dogs & peanuts
>>> intersect
{'sally', 'linus'}

>>> diff = cs134_dogs - peanuts
>>> diff
{'chelsea', 'artie', 'wally', 'pixel'}

>>> cs134_dogs
{'sally', 'wally', 'linus', 'artie', 'chelsea', 'pixel'}

Original set is unchanged!
```
Set Operations: Mutators

```python
>>> cs134_dogs = {"wally", "pixel", "linus", "chelsea", "sally", "artie"}
>>> peanuts = {"sally", "linus", "charlie", "franklin", "lucy", "patty"}

>>> cs134_dogs |= peanuts
Original set is mutated!
{'sally', 'wally', 'patty', 'chelsea', 'pixel', 'franklin', 'lucy', 'artie', 'linus', 'charlie'}

>>> cs134_dogs &= peanuts
Original set is mutated!
{'sally', 'linus'}

>>> cs134_dogs -= peanuts
Original set is mutated!
{'wally', 'artie', 'chelsea', 'pixel'}
```
Set Operations

- The usual operations you think of in set theory are implemented as follows.

The following operations always return a **new set**.

- **s1 | s2 (Set Union)**
  - Returns a new set that has all elements that are either in s1 or s2.

- **s1 & s2 (Set Intersection)**
  - Returns a new set that has all the elements that are common to both sets.

- **s1 - s2 (Set Difference)**
  - Returns a new set that has all the elements of s1 that are not in s2.

- **s1 |= s2, s1 &= s2, s1 -= s2** are versions of |, &, - that mutate s1 to become the result of the operation on the two sets.
Set Examples
(live coding)
voting.py
Tuple Examples
(live coding)

madlibs.py