CS134 Lecture 14:
Tuples and Sets
Announcements & Logistics

- **HW 6** will be released today and due Mon @ 10 pm
  - Short HW (only 5 questions)
  - Covers topics this week (mutability, aliasing, scope, tuples, sets)
- **Lab 4 Part 2** due Wednesday/Thursday 10pm
- **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 HW 6

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

- **Scope**: variables, functions, objects have limited accessibility/visibility.
  - Understanding how this works helps us make decisions about where to define variables/functions/objects

Goal was to demystify surprising behavior: nothing in computer science is magic!
Today's Plan

• 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)
def my_func(lst):
    lst.append(1)  # same effect as lst += [1]
    print('local lst', lst)
    return lst

lst = [3]
ew_lst = my_func(lst)
print('global lst', lst)
print('new_lst', new_lst)

>>> python3 example.py
def my_func(lst):
    lst.append(1)  # same effect as lst += [1]
    print('local lst', lst)
    return lst

lst = [3]
new_lst = my_func(lst)
print('global lst', lst)
print('new_lst', new_lst)

>>> python3 example.py

Review: Scope Example from Lecture 4
def my_func(lst):
    lst.append(1)  # same effect as lst += [1]
    print('local lst', lst)
    return lst

lst = [3]
new_lst = my_func(lst)
print('global lst', lst)
print('new_lst', new_lst)

Review: Scope Example from Lecture 4

>>> python3 example.py
Review: Scope Example from Lecture 4

```python
def my_func(lst):
    lst.append(1)  # same effect as lst += [1]
    print('local lst', lst)
    return lst

lst = [3]
ew_lst = my_func(lst)
print('global lst', lst)
print('new_lst', new_lst)
```

```shell
>>> python3 example.py
```

Diagram showing the scope of `lst` and `my_func()`.
def my_func(lst):
    lst.append(1)  # same effect as lst += [1]
    print('local lst', lst)
    return lst

lst = [3]
ew_lst = my_func(lst)
print('global lst', lst)
print('new_lst', new_lst)

>>> python3 example.py
local lst [3, 1]
def my_func(lst):
    lst.append(1)  # same effect as lst += [1]
    print('local lst', lst)
    return lst

lst = [3]
ew_lst = my_func(lst)
print('global lst', lst)
print('new_lst', new_lst)

>>> python3 example.py
local lst [3, 1]
def my_func(lst):
    lst.append(1)  # same effect as lst += [1]
    print('local lst', lst)
    return lst

lst = [3]
ew_lst = my_func(lst)
print('global lst', lst)
print('new_lst', new_lst)

>>> python3 example.py
local lst [3, 1]
```python
def my_func(lst):
    lst.append(1)  # same effect as lst += [1]
    print('local lst', lst)
    return lst

lst = [3]
new_lst = my_func(lst)
print('global lst', lst)
print('new_lst', new_lst)
```

```
>>> python3 example.py
local lst [3, 1]
global lst [3, 1]
```
def my_func(lst):
    lst.append(1)  # same effect as lst += [1]
    print('local lst', lst)
    return lst

lst = [3]
ew_lst = my_func(lst)
print('global lst', lst)
print('new_lst', new_lst)

>>> python3 example.py
Local lst [3, 1]
Global lst [3, 1]
New_lst [3, 1]
def my_func(lst):
    lst.append(1)  # same effect as lst += [1]
    print('local lst', lst)
    return lst

lst = [3]
ew_lst = my_func(lst)
print('global lst', lst)
print('new_lst', new_lst)

$ python3 example.py
local lst [3, 1]  
global lst [3, 1]  
new_lst [3, 1]   
$
Aliasing and Scope

• When we pass a mutable object as a parameter to a function, within the function, that local parameter variable is an alias

• Since a list is mutable, changes to the alias affect the original!

• When we pass an immutable object as a parameter to a function, within the function, that local parameter variable is a clone

• Wouldn't it be nice to have an immutable form of a list?
Tuples
Tuples: An Immutable Sequence

- Tuples are an **immutable sequence of values** (almost like immutable lists) separated by commas and enclosed within parentheses ()

```
# string tuple
>>> names = ("Bill", "Lida", "Shikha")

# int tuple
>>> primes = (2, 3, 5, 7, 11)

# singleton
>>> num = (5,)

# parentheses are optional
>>> values = 5, 6

# empty tuple
>>> emp = ()
```

A tuple of size one is called a **singleton**. Note the (funky) syntax.
Tuples as Immutable Sequences

- Tuples, like strings, support any sequence operation that does not involve mutation: e.g,
  - `len()` function: returns number of elements in tuple
  - `[]` indexing: access specific element
  - `+`, `*`: tuple concatenation
  - `[:`]: slicing to return subset of a tuple (as a new tuple)
  - `in` and `not in`: check membership of an object in a tuple
  - `for-loops`: iterate over elements in tuple
Multiple Assignment and Unpacking

- Tuples support a simple syntax for assigning multiple values at once, and also for "unpacking" sequence values

```python
>>> a, b = 4, 7  # after evaluating: a == 4, b == 7
# reverse the order of values in tuple
>>> b, a = a, b
# tuple assignment to "unpack" list elements
>>> cb_info = ['Charlie Brown', 8, False]
>>> name, age, glasses = cb_info
```

- Note that the preceding line is just a more compact way of writing:

```python
>>> name = cb_info[0]
>>> age = cb_info[1]
>>> glasses = cb_info[2]
```
Multiple Return from Functions

- Tuples come in handy when returning multiple values from functions

```python
# multiple return values as a tuple
def arithmetic(num1, num2):
    '''Takes two numbers and returns their sum and product'''
    return num1 + num2, num1 * num2

>>> arithmetic(10, 2)
(12, 20)

>>> type(arithmetic(3, 4))
<class 'tuple'>
```
Conversion between Sequences

• The functions `tuple()`, `list()`, and `str()` convert between sequences

```python
>>> word = "Williamstown"
>>> char_lst = list(word) # string to list
>>> char_lst
['W', 'i', 'l', 'l', 'i', 'a', 'm', 's', 't', 'o', 'w', 'n']
>>> char_tuple = tuple(char_lst) # list to tuple
>>> char_tuple
('W', 'i', 'l', 'l', 'i', 'a', 'm', 's', 't', 'o', 'w', 'n')
>>> list((1, 2, 3, 4, 5)) # tuple to list
[1, 2, 3, 4, 5]
```
Conversion between Sequences

- The functions `tuple()`, `list()`, and `str()` convert between sequences

```python
>>> str(("hello", "world"))  # tuple to string
"('hello', 'world')"

>>> num_range = range(12)

>>> list(num_range)  # range to list
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]

>>> str(list(num_range))  # range to list to string
'\n[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]'
Takeaways

• **Tuples** are a new *immutable* sequence that:
  • support all sequence operations such as indexing and slicing
  • are useful for argument unpacking, multiple assignments
  • are useful for handling list-like data without aliasing issues
Sets
New Unordered Data Structure: Sets

- Lists are **ordered** collections of objects
- What if we only need an unordered collection of individual items?
  - We can use a new 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 with sets?

```python
>>> first_choice = {'a', 'b', 'a', 'a', 'b', 'c'}
>>> uniques = set(first_choice)
>>> uniques
# ???
>>> set("aabraakadabra")
# ???
```
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
----> 1  {[3, 2], [1, 5, 4]}
```

```
TypeError: unhashable type: 'list'
```
Set Operations

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

  The following operations always return a new set.

- \( s_1 | s_2 \) (Set Union)
  - Returns a new set that has all elements that are either in \( s_1 \) or \( s_2 \)

- \( s_1 \ & \ s_2 \) (Set Intersection)
  - Returns a new set that has all the elements that are common to both sets.

- \( s_1 - s_2 \) (Set Difference)
  - Returns a new set that has all the elements of \( s_1 \) that are not in \( s_2 \)

- \( s_1 |= s_2, s_1 &= s_2, s_1 -= s_2 \) are versions of |, &, - that mutate \( s_1 \) 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  # Original set is unchanged!
{'sally', 'wally', 'linus', 'artie', 'chelsea', 'pixel'}
```
Set Operations: Mutators

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

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

```python
>>> cs134_dogs = {"wally", "pixel", "linus", "chelsea", "sally", "artie"}
>>> cs134_dogs &= peanuts
>>> cs134_dogs  # Original set is mutated!
{'sally', 'linus'}
```

```python
>>> cs134_dogs = {"wally", "pixel", "linus", "chelsea", "sally", "artie"}
>>> cs134_dogs -= peanuts
>>> cs134_dogs  # 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.

• \( s_1 \lor s_2 \) (Set Union)
  • Returns a new set that has all elements that are either in \( s_1 \) or \( s_2 \)

• \( s_1 \land s_2 \) (Set Intersection)
  • Returns a new set that has all the elements that are common to both sets.

• \( s_1 - s_2 \) (Set Difference)
  • Returns a new set that has all the elements of \( s_1 \) that are not in \( s_2 \)

• \( s_1 \mid= s_2, s_1 \&= s_2, s_1 -= s_2 \) are versions of \( \lor, \land, - \) that mutate \( s_1 \) to become the result of the operation on the two sets.
Set Examples
(live coding)