Tuples, Sets, and Frozen Sets

Thomas Schwarz, SJ

Tuples

- Tuples are like *immutable* lists.
  - They are immutable, i.e. you cannot change them once they have been created.
  - This allows us to use them as keys for a dictionary

Tuple Creation

- You create a tuple by putting a comma separated list of items in parentheses

```
small_primes = (2,3,5,7,11,13)
```

```
digits = ("0", "1", "2", "3", "4", "5", "6", "7", "8", “9")
```

Accessing Elements

- You access tuple coordinates by using the same notation as for lists

```
digits = ("0", "1", "2", "3", "4", "5", "6", "7", "8", "9"
print(digits[5])
```

- prints out “5”

Using Tuples: Tuple Assignment

- Tuple assignment
  - The “tuple operator” is the comma
    - Meaning, putting commas between things creates a tuple
  - Tuples can be assigned

Using Tuples: Tuple Assignment

- Tuple assignment
  - The “tuple operator” is the comma
    - Meaning, putting commas between things creates a tuple
    - Tuples can be assigned as tuples
    - Which assigns the elements of the tuple as well
- Example:

```
a, b = 3, 5
```

  - Creates two tuples and makes them equal
  - Result is a is 3 and b is 5

Using Tuples: Tuple Assignment

- Tuple assignment makes it easy to switch values
  - Assume that we have two variables
  - We want them to exchange values
  - Here is code that does not succeed:

```
a=3
b=5

#now we want to switch values
a=b
b=a
print(a,b)  #prints 5 5
```

- Spend some time figuring out why

Using Tuples: Tuple Assignment

- When we assign b=a, the old value of a has just be overwritten

```
a=3
b=5

#now we want to switch values
a=b
b=a
print(a,b)  #prints 5 5
```

Using Tuples: Tuple Assignment

- We need to safeguard the value of *a* in a temporary variable
  - This is a well-known trap for beginners
  - But now we have three assignments

```
a=3
b=5

#now we want to switch values
temp = a
a=b
b=temp
print(a,b)  #prints 5 3
```

Using Tuples: Tuple Assignment

- With tuples, this works much simpler

```
a=3
b=5

#now we want to switch values
a,b = b,a
print(a,b)  #prints 5 3
```

- The right side of the assignment is a tuple
- We assign it as a tuple to the left side
- Which then updates the values of a and b

Using Tuples: Unpacking

- In general, you can *unpack* a tuple through an assignment
  - On the left, you have a tuple with variables
  - On the right, you have an established tuple

```
(name, last_name, birth_year, birth_month, birth_date) = caesar
```

  - This will load name, last_name, birth_year, … with the values in caesar
  - The number of elements on both sides of the assignment needs to be the same

Using Tuples: Unpacking

- You can even unpack when calling a function
  - Put an asterisk before the tuple to cause the unpacking
    - Define a function of two variables

```
def geo_mean(a,b):
    return (a*b)**(1/2)
```

- We call it in the usual way

```
print(geo_mean(4,7))
```

- But we can also call it with a tuple

```
tp = (3,7)
print(geo_mean(*tp))
```

Using Tuples: Several Return Values

- Assume that you want to return more than one value from a function
  - You can “kludge” it by return a list
    - Then you access the various return values via indices
- You can return a tuple
  - And use tuple unpacking at the other end

Using Tuples: Unpacking

- Several return values example
  - Assume that you want to return the mean and the standard deviation of a list of numbers

```
import math
def stats(lista):
    if not lista:          #lista is empty
        return 0,0
    mean = 0
    var = 0
    for element in lista:
        mean += element
    for element in lista:
        var += (element-mean)**2
    return mean/len(lista), math.sqrt(var/len(lista))
```

Using Tuples: Unpacking

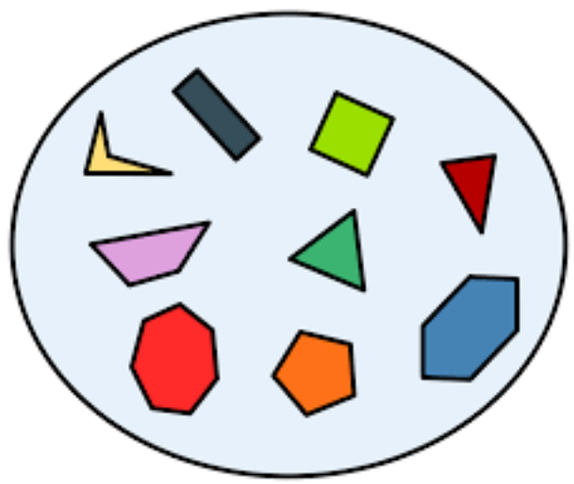
- This code returns a tuple

```
def stats(lista):
        …
        return mean/len(lista), math.sqrt(var/len(lista))
```

- If we call this function, we unpack in a single statement

```
mu, sigma = stats([12,23,12,12,14,12,13,16,29,11,12,13])
```

SETS



Sets

- Sets are unordered, iterable and mutable
  - You can use a for loop on a set: for x in A:
  - You can add and delete elements from a set
    - Using the add and remove methods
- You define a set through the set keyword or by writing it in curly brackets

Set Example

- Determine all the symbols in a string not in another
  - Easiest with set notation
  - Create a set for each string
  - Use the set operation minus to get all elements in the first set that are not in the second
  - Return as a string
    - Notice, sets are *iterable*
      - This means that we can systematically walk through a set, e.g. with a for loop

Set Example

```
def minus(string1, string2):
    my_set = set(string1)
    my_other = set(string2)
    return "".join(my_set - my_other)
```

Create sets from the strings

Return the result as a string

Use the minus operation

```
>>> minus("adsfijroiupoqewiurp", "qwroiupsadf")
'je'
```

Frozen Sets

- Sets are mutable, so they cannot be keys for a dictionary
- If you want sets to be the keys in a dictionary, use the frozen set instead.