

CSC 241 Notes
Yosef Mendelsohn

With many thanks to Dr. Amber Settle for the original version of these notes.

Prof Settle's notes in turn are based on the Perkovic text.

# The set collection type

The set type is a collection type used to store an unordered collection of immutable values.

One use of sets is in removing duplicates from sequences since sets do not contain duplicates by definition.

Consider the following examples:

>>> grains = {'rice', 'wheat', 'corn', 'rye', 'oat', 'wheat', 'millet'}

>>> grains

{'wheat', 'corn', 'rye', 'millet', 'oat', 'rice'}

>>> type(grains)

<class 'set'>

>>> fruit = {}

>>> type(fruit)

<class 'dict'>

>>> fruit = set()

>>> fruit

set()

>>> fruit.add("apple")

>>> fruit

{'apple'}

## Set problems

Do the following **CodeLab exercises** in the Week 8 folder:

* 51800
* 51802
* 51803
* 51804
* 51805
* 51806
* 51808
* 51809
* 51810

**Problem**: Recall the function we wrote in the dictionary lecture:  duplicates() that takes as a parameter a string representing the name of a text file.  The function *returns* True if the file contains duplicate words and False if the file doesn't contain duplicate words.  Make sure to remove all punctuation (i.e. all commas, question marks, periods, colons, and semicolons) before determining whether any words are duplicated so that the punctuation won't interfere with the process.  Your function should ignore case (e.g. APPLE = Apple = apple).

The above version used a dictionary to determine if there were duplicates. Rewrite the function as duplicates\_set() so that it uses a set instead of a dictionary. (It should be much simpler).

See the code in the solutions file for this lecture.

# The tuple collection type

The tuple type is the same as a list except that **a tuple is immutable**, i.e. a tuple cannot be modified.

You'll also note that we create a tuple using parentheses as opposed to square brackets.

>>> t = (3,-7)

>>> t

(3, -7)

>>> type(t)

<class 'tuple'>

>>> t[0]

3

>>> t[1] = 3 *🡪 trying to modify the tuple!*

Traceback (most recent call last):

 File "<pyshell#4>", line 1, in <module>

 t[1] = 3

TypeError: 'tuple' object does not support item assignment

**Collection type properties** for tuples include:

* Items in the collection are ordered
* Items in the collection are accessed using an index (offset)
* Like strings – but unlike lists -- tuples are immutable
* Tuples have a fixed length, since they cannot change. For example, whereas lists can grow and shrink, tuples can not.

Because tuple objects are immutable, a tuple can be used as a dictionary **key**.

More about class tuple

* <https://docs.python.org/3/library/stdtypes.html#tuples>
* >>> help(tuple)

## Tuple problems

Do the following **CodeLab exercises** in the Week 8 folder:

* 51701
* 51702
* 51703
* 51704
* 51705
* 51706
* 51707

**Problem**: Implement a function **lookup**() that provides a phonebook lookup feature.

The function takes as a parameter a dictionary representing a phonebook. In the dictionary, tuples containing first and last names of individuals (the keys) are mapped to strings containing phone numbers (the values).

Your function should provide a simple user interface through which a user can enter the first and last name of an individual and obtain the phone number assigned to that individual. It should indefinitely prompt the user for first and last names, stopping only when the user does a keyboard interrupt (e.g. control-c).

For example, it would be used as follows:

>>> phonebook = {('Luca', 'Elam'): '(312) 123-4567',\

 ('Djengo', 'Settle'): '(773) 987-6543',\

 ('Devon', 'Reilly'): '(520) 454-6677'}

>>> lookup(phonebook)

Enter the first name: Luca

Enter the last name: Elam

(312) 123-4567

Enter the first name: Devon

Enter the last name: Reilly

(520) 454-6677

Enter the first name:

See the solution in the solutions file for this week.

# The random module

Random numbers are useful in a variety of contexts:

* Computer games (e.g. poker, blackjack, dice games, action games, etc)
* Simulations for science, engineering, and finance
* Statistical analysis
* Computer security protocols

Unfortunately the machines we use are deterministic, which means (among other things) they have no access to 'perfect' randomness.

Instead we have to make do with **pseudo random numbers**, i.e. “almost” random numbers.

A pseudo random number generator produces a sequence of numbers that “look” random. These generators are typically good enough for most applications that need random numbers. For certain very subtle or high-performance situations where "true" randomness is needed, other techniques must b eemployed.

To use a pseudo random number generator in Python, we need to import the random module:

>>> import random

## Functions in the random module

There are a number of functions in the random module that are helpful.

We’ll work through a number of them.

### Choosing a random integer

The function **randrange**() takes a pair of integers and returns some number in the range [a, b), that is including a but not including b.

>>> random.randrange(1,7)

3

>>> random.randrange(1,7)

4

>>> random.randrange(1,7)

6

>>> random.randrange(1,7)

6

>>> random.randrange(1,7)

3

>>> random.randrange(1,7)

2

There is a very similar function called **randint**() takes a pair of integers and returns some number in the range [a, b]. That is, whereas randrange() returns a value up to but not including b, randint() can include b.

randrange() also provides an additional parameter 'step' – we won't go into it here, but feel free to read up on it.

### Choosing a random decimal

Sometimes we need a decimal random number rather than a decimal integer.

In that case we need to use the **uniform()** function which takes two numbers a and b and returns a float number x such that a ≤ x ≤ b (assuming that a ≤ b), with each float in the range equally likely.

The following obtains several random numbers between 0 and 1:

>>> random.uniform(0,1)

0.7378384739678002

>>> random.uniform(0,1)

0.09911455205729514

>>> random.uniform(0,1)

0.18295866524385507

>>> random.uniform(0,1)

0.3319868023085931

### Shuffling, choosing and sampling

The function **shuffle**() shuffles, or permutes, the objects in a Python sequence (e.g. a list, set, etc) – much like the way a deck of cards is shuffled before playing a card game.

>>> lst = [1, 2, 3, 4, 5]

>>> random.shuffle(lst)

>>> lst

[1, 2, 4, 5, 3]

>>> random.shuffle(lst)

>>> lst

[3, 5, 2, 1, 4]

>>> words = ['cat', 'bat', 'at', 'fat']

>>> random.shuffle(words)

>>> words

['at', 'cat', 'bat', 'fat']

The function **choice**() allows us to choose an item from a container uniformly at random:

>>> random.choice(lst)

5

>>> random.choice(lst)

4

>>> random.choice(lst)

1

>>> random.choice(words)

'fat'

>>> random.choice(words)

'cat'

>>> random.choice(words)

'cat'

>>> random.choice(words)

'fat'

We use the **sample**() function if instead of choosing a single item from a sequence we wanted to choose a sample of size k, with every sample equally likely. The function takes the container and the number k as parameters:

>>> random.sample(lst, 2)

[1, 2]

>>> random.sample(lst, 2)

[4, 2]

>>> random.sample(lst, 3)

[3, 4, 1]

>>> random.sample(words, 3)

['at', 'cat', 'bat']

>>> random.sample(words, 3)

['fat', 'at', 'cat']

>>> random.sample(words, 2)

['fat', 'bat']