# Lab 5: Dictionaries and Sets

CSE/IT 107

NMT Department of Computer Science and Engineering

---

"All thought is a kind of computation."

— D. Hobbes

"Vague and nebulous is the beginning of all things, but not their end."

— K. Gibran

"It [programming] is the only job I can think of where I get to be both an engineer and artist. There's an incredible, rigorous, technical element to it, which I like because you have to do very precise thinking. On the other hand, it has a wildly creative side where the boundaries of imagination are the only real limitation."

— A. Hertzfeld

---

## 1 Introduction

In previous labs, we have used lists and tuples to be able to enclose data in a certain structure and manipulate it. Lists gave us an easy way to find sums of numbers, to store things, to sort things, and much more. Structures like this are commonly referred to as collections: collections collect data and encapsulate it. They give us useful methods to manipulate that data.

A list in Python is an ordered container of any elements you want indexed by whole numbers starting at 0. Lists are mutable: this means you can add elements, change elements, and remove elements from a list. Meanwhile, tuples are immuatable and cannot be changed once they are created. In this lab, we will introduce you to two other collections: Sets and dictionaries.

## Contents

## 2 Sets

Sets are a lot like lists, but their properties are a bit different. A set is just like a list, but *no element can appear twice* and it is *unordered*. Additionally, they cannot contain mutable elements. Thus, a set cannot contain a list. Sets themselves are mutable.

A set is made using curly braces or from a list:

```
>>> a = {5, 5, 4, 3, 2} # duplicates ignored
>>> print(a)
{2, 3, 4, 5}
>>> b = set([5, 5, 4, 3])
>>> print(b)
{3, 4, 5}
```

Sets are heterogeneous – their elements do not need to be of the same type:

```
>>> a = {5, 4, 'a'}
>>> print(a)
{'a', 4, 5}
```

Because sets are inherently unordered, they cannot be indexed. That means you cannot use the [] operator on sets:

```
>>> a = {5, 4, 3, 2}
>>> a[1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'set' object does not support indexing
```

The | operator can be used to join sets together. This results in a new set composed of the elements of the two composite sets. This is called the *union* of the two sets. Since the result is also a set, any duplicate values will be only included once.

```
>>> a = {1, 2, 4, 8, 16}
>>> b = {1, 3, 9, 27}
>>> a | b
{1, 2, 3, 4, 8, 9, 16, 27}
```

The & operator can be used to find the intersection of two sets. This is the set containing all elements that are in both sets.

```
>>> a = {2, 4, 6, 8, 10, 12}
>>> b = {3, 6, 9, 12, 15, 18}
>>> a & b
{12, 6}
```

The - operator can be used to find the difference of two sets. This is the first set with all of the elements of the second set removed.

```
>>> a = {2, 4, 6, 8, 10, 12}
>>> b = {3, 6, 9, 12, 15, 18}
>>> a - b
{8, 2, 10, 4}
```

The ^ operator can be used to find the symmetric difference of two sets. This is the set containing all elements in one of the original sets, but not in both.

```
>>> a = {'one', 'eight'}
>>> b = {'two', 'four', 'six', 'eight'}
>>> a ^ b
{'one', 'six', 'two', 'four'}
```

A list can be converted into a set using the `set()` function:

```
>>> a = [1, 5, 2, 'hello', 2.3, 5, 2]
>>> set(a) # duplicate removed!
{1, 2, 'hello', 2.3, 5}
```

Similarly, a set can be converted into a list using the `list()` function:

```
>>> a = {1, 2, 2, 5, 'asdf', 3, 2}
>>> list(a)
[1, 2, 3, 5, 'asdf']
```

You may also iterate over a set and add elements using the `.add()` method:

```
>>> companions_nine = {'rose', 'jack'}
>>> companions_ten = {'rose', 'mickey', 'jack', 'donna', 'martha', 'wilf'}
>>> # iterating over elements
... for companion in companions_nine:
...     print(companion)
...
rose
jack
>>> # adding an element to set
... companions_ten.add('sarah jane')
>>> companions_ten
{'donna', 'sarah jane', 'wilf', 'rose', 'martha', 'mickey', 'jack'}
```

## 2.1 Summary

- Sets can be made using curly braces or using the `set()` function given another sequence type:

  ```
  food = {'burgers', 'carne adovada', 'burritos'}
  food = set(['burgers', 'carne adovada', 'burritos'])
  food = set(('burgers', 'carne adovada', 'burritos'))
  ```

  *Note that empty curly braces create an empty dictionary.* To create an empty set, use the `set()` function:

  ```
  >>> type({})
  <class 'dict'>
  >>> type(set())
  <class 'set'>
  ```

- No element can appear twice.
- Can only contain immutable elements (for example, lists are not allowed as elements, but tuples are as long as the tuple only contains immutable elements).
- Given two sets `a` and `b`:

  | | |
  |---|---|
  | `a \| b` | union of `a` and `b` |
  | `a & b` | intersection of `a` and `b` |
  | `a - b` | difference of `a` and `b` |
  | `a ^ b` | symmetric difference of `a` and `b` |

- Python docs:

  `https://docs.python.org/3.3/tutorial/datastructures.html#sets`

## 3 Dictionaries

Dictionaries are similar to lists, but with a different index system. The elements of a dictionary are stored using a key-value system. A key is used similarly to the indices of a list, but it must be of any immutable type. A value is simply the element associated with that key and can be anything. This means that dictionaries are useful for storing elements best identified by a description or name rather than a strict order.

- A dictionary is made using curly braces, followed by listing the key-value pairs that make up the dictionary. Keys and values are separated by colons, while key-value pairs are separated by commas.
- Values can be accessed, changed, or added by using their keys just as you would use the indices of a list.

```
>>> food = {'breakfast': 'burrito', 'lunch': 'burger', 'dinner': 'tacos'}
>>> print(food)
{'dinner': 'tacos', 'lunch': 'burger', 'breakfast': 'burrito'}
>>> food['breakfast'] = 'toast'
>>> print(food)
{'dinner': 'tacos', 'lunch': 'burger', 'breakfast': 'toast'}
>>> food['brunch'] = 'biscuits'
>>> print(food)
{'dinner': 'tacos', 'brunch': 'biscuits', 'lunch': 'burger', 'breakfast': 'toast'}
```

- Like sets, the elements of a dictionary are unordered.
- A list of the keys of a dictionary can be accessed using the `.keys()` function.

```
>>> a = {1: 5, 'cake': 'lots', 'color': 'green'}
>>> print(a)
{1: 5, 'cake': 'lots', 'color': 'green'}
>>> print(a.keys())
dict_keys([1, 'cake', 'color'])
```

- Only immutable elements can be used as keys.
- (Actually, only *hashable* elements can be used as keys; however, this usually means immutable. The error message will tell you that a type is *unhashable* when you used a mutable element as a key or set element.)
- Since tuples are immutable, they can be used as keys for dictionaries. However, the tuple must not contain anything mutable, like a list, dictionary, or set.

```
>>> a = {}
>>> a[('a', 'b')] = 5
>>> print(a)
{('a', 'b'): 5}
>>> a[('a', 'c')] = 6
>>> print(a)
{('a', 'c'): 6, ('a', 'b'): 5}
>>> a['a', []] = 7
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

- You can use the `in` keyword to test whether an item is a key in the respective dictionary.
- You can use a for loop to iterate over the *keys* of a dictionary.
- `.items()` returns a list of tuples where the first element is the key of a dictionary item and the second element is the corresponding value. This can be used in a for loop as well with the unpacking syntax.

```
>>> states = {'NM' : 'New Mexico', 'TX' : 'Texas', 'KS' : 'Kansas'}
>>> 'NM' in states
True
>>> for state_short in states:
...     print(state_short, states[state_short])
...
KS Kansas
TX Texas
NM New Mexico
>>> states.items()
dict_items([('KS', 'Kansas'), ('TX', 'Texas'), ('NM', 'New Mexico')])
>>> for state_short, state_name in states.items():
...     print(state_short, state_name)
...
KS Kansas
TX Texas
NM New Mexico
```

Dictionaries are good for a load of things. Here's an example:

```
states = {'NM' : 'New Mexico', 'TX' : 'Texas', 'KS' : 'Kansas'}
capitals = { # multiline dictionaries are easier to read
  'NM' : 'Santa Fe',
  'TX' : 'Austin',
  'KS' : 'Kansas City',
}

state = input('Enter a state >>> ')
if state in states:
    print('You selected {}. The state capital is {}.'.format(states[state],
        capitals[state]))
else:
    print('The state you selected is not known to this program.')
```

## 4 Collection Similarities

Collections share several properties that make them convenient to work with.

- The `len()` function can be used on any collection object in order to find the number of elements.
- All collections can be iterated over with `for` loops, getting one element with each iteration. For dictionaries, the values provided will be the keys. Note that for sets and dictionaries the order is not fixed.

```
>>> a = {'a': 1, 'b': 2}
>>> for i in a:
...     print(i)
...     print(a[i])
...
b
2
a
1
```

- All collections can use the `in` keyword to test if an element is in that collection. For dictionaries, this will compare against the list of keys, not the values.

```
>>> a = {'1': 1}
>>> 1 in a
False
>>> '1' in a
True
```

- Tuples, sets, and lists can all be freely converted from one to another using the `tuple()`, `set()`, `list()`, `str()`, and `dict()` functions.

## 5 Collections Summary

| Data Structure | Mutable | Mutable elements | Indexing | Ordered | Other properties |
|---|---|---|---|---|---|
| List `[]` | yes | yes | by whole numbers | yes | can contain elements more than once |
| Sets `{}` | yes | no | not indexable | no | no element can appear more than once |
| Tuples `()` | no | yes | by whole numbers | yes | can contain elements more than once |
| Dictionary `{}` | yes | yes | by anything "hash-able" (immutable collections or basic types) | no | |

**Table 1:** Summary of Data Structures in Python

## 6 Stacks

Stacks are an important data structure in the world of computer science. They are at the heart of every operating system and used in many, many pieces of software.

For a stack, imagine a stack of plates. You can only add plates to the top and remove plates from the top. We call this a "Last-In-First-Out" data structure: If you add three plates to your stack, the last one you added will be the first one you remove.

Similar to that, in Python you can say that a stack is a list where you can only add elements to the end or remove elements from the end. In Python, this is accomplished using the `.pop()` and `.append(element)` methods on lists. When given no arguments, the pop method will remove the last element of the list and return it. The append method will add an element to the end of the list.

When we ask you to use a stack in Python, you should use a list and only use these two methods and the array index `[-1]` to inspect the last element of the list. For example:

```
>>> somelist = [1, 2, 3]
>>> a = somelist.pop()
>>> print(a)
3
>>> print(somelist)
[1, 2]
>>> somelist.append(4)
>>> print(somelist)
[1, 2, 4]
>>> somelist[-1]
4
```

Interacting with the elements of the list in any other way violates the idea of the list being a stack.

## 7 Exercises

**Requirements**

Remember PEP 8, PEP 257, and the boilerplate code requirement.

**Exercise 7.1** (anagrams.py).
Write a function that takes in two strings. If the strings are anagrams of one another, return `True`. If not, return `False`.

Two strings are anagrams of one another if the characters of one can be rearranged to make the other.

**Exercise 7.2** (days.py).
Write a function that takes four arguments – day, month, and year as numbers, and weekday as MTWRFSU – and converts this date to a human-readable string. Have the program be called with user-specified input. For example:

```
Enter day >>> 28
Enter month >>> 9
Enter year >>> 2014
Enter weekday >>> U
Date is: Sunday, September 28, 2014
```

Use dictionaries to convert weekdays to their long name and months to their long name.

**Exercise 7.3** (rpn_calculator.py).
Write a reverse Polish notation calculator. In reverse Polish notation (also known as HP calculator notation), mathematical expressions are written with the operator following its operands. For example, $3+4$ becomes $3\ 4\ +$.

Order of operations is entirely based on the ordering of the operators and operands. To write $3+(4*2)$, we would write $4\ 2\ *\ 3\ +$ in RPN. The expressions are evaluated from left to right.

A longer example of an expression is this:

$$5\ 1\ 2\ /\ 4\ *\ +\ 3\ -$$

which translates to

$$5+((1/2)*4)-3$$

If you were to try to "parse" the RPN expression from left to right, you would probably "remember" values until you hit an operator, at which point you take the last two values and use the operator on them. In the example expression above, you would store 5, then 1, then 2, then see the division operator (/) and take the last two values you stored (1 and 2) to do the division. Then, you would store the result of that (0.5) and encounter 4, which you store. When you encounter the multiplication sign (*), you would take the last two values stored and do the operation $(4*0.5)$ and store that.

Following this through step by step, the steps would look something like this (the bold number is the most recently computed value):

$$5\ 1\ 2\ /\ 4\ *\ +\ 3\ -$$

$$5\ \mathbf{0.5}\ 4\ *\ +\ 3\ -$$

$$5\ \mathbf{2}\ +\ 3\ -$$

$$\mathbf{7}\ 3\ -$$

$$\mathbf{4}$$

Writing this algorithm for evaluating RPN in pseudo code, we get:

```
1. Read next value in expression.
2. If number, store.
3. If operator:
    (a) Remove last two numbers stored.
    (b) Do operation with these last two numbers.
    (c) Store the result of the operation as last number.
```

If you keep repeating this algorithm, you will eventually just end up with one number stored unless the RPN expression was invalid.

Your task is to write an RPN calculator which asks the user for an RPN expression and prints the result of that expression. You *must* use a stack (see Section 6). The RPN algorithm has to be in a separate function (not main). You need to support the four basic operators (+, -, *, and /).

You should detect and display messages for the following errors:

- Operand is used when not enough numbers are stored.
- More or less than one number stored after the expression is evaluated.

Please see the example input and output below for expressions you can test with.

| RPN Expression | Output |
|---|---|
| 5 1 2 / 4 * + 3 − | 4 |
| 4 2 + 1 5 + * + | Not enough operands for +. |
| 2 100 3 * 5 + 2 2 2 + + * * | 3660 |

**Exercise 7.4** (lettercount.py).

Write program that reads in a string on the command line and returns a table of the letters of the alphabet in alphabetical order which occur in the string together with the number of times each letter occurs. Case should be ignored. A sample run of the program would look this:

```
Enter some letters >>> The cat in the hat
a 2
c 1
e 2
h 3
i 1
n 1
t 4
```

This should involve writing a function that takes in a string and returns a dictionary with these letters and counts.

## 8 Submitting

You should submit your code as a tarball. It should contain all files used in the exercises for this lab. The submitted file should be named

```
cse107_firstname_lastname_lab5.tar.gz
```

**Upload your tarball to Canvas.**

## List of Files to Submit