Lab 4 Summary

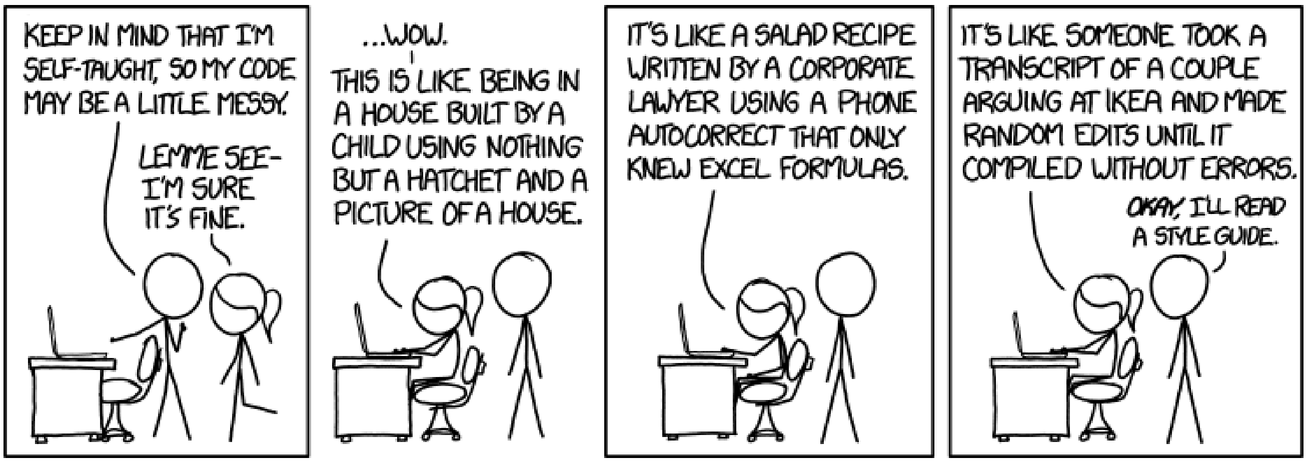
Data Structures and Debugging

▶ Study *Think Python* Chapter 11: Dictionaries

▶ Study *Think Python* Chapter 12: Tuples

▶ Study *Think Python* Appendix A: Debugging

1. Python Coding Style


KEEP IN MIND THAT I'M SELF-TAUGHT, SO MY CODE MAY BE A LITTLE MESSY.
LEMME SEE– I'M SURE IT'S FINE.
...WOW.
THIS IS LIKE BEING IN A HOUSE BUILT BY A CHILD USING NOTHING BUT A HATCHET AND A PICTURE OF A HOUSE.
IT'S LIKE A SALAD RECIPE WRITTEN BY A CORPORATE LAWYER USING A PHONE AUTOCORRECT THAT ONLY KNEW EXCEL FORMULAS.
IT'S LIKE SOMEONE TOOK A TRANSCRIPT OF A COUPLE ARGUING AT IKEA AND MADE RANDOM EDITS UNTIL IT COMPILED WITHOUT ERRORS.
OKAY, I'LL READ A STYLE GUIDE.

https://xkcd.com/1513/

Python programmers follow the style guide described in PEP 8 which make it easy to read Python code and therefore it is much easier to understand and spot mistakes (see http://www.python.org/dev/peps/pep-0008/).

- Use 4-space indentation, and no tabs. Four spaces are a good compromise between small indentation (allows greater nesting depth) and large indentation (easier to read). Tabs introduce confusion and are best left out.

- Wrap lines so that they don't exceed 79 characters. This helps users with small displays and makes it possible to have several code files side-by-side on larger displays.
- Use blank lines to separate functions and classes, and larger blocks of code inside functions.
- When possible, put comments on a line of their own.
- Use docstrings.
- Use spaces around operators and after commas, but not directly inside bracketing constructs: a = f(1, 2) + g(3, 4).
- Name your classes and functions consistently. The convention is to use CamelCase for classes and snake_case for functions, methods and variables.

2. Storing data using Sets

We have seen how to store collections of data using *lists*. A list uses square brackets [ ].

Three other useful kinds of collections are *sets*, *tuples*, and *dictionaries*. A set uses curly brackets { }, a tuple uses round brackets ( ) and a dictionary also uses curly brackets { }.

With four different options for storing your collections of data, you will be able to pick the one that best matches your problem in order to keep your code as clean and efficient as possible.

Sets

A *set* is an unordered collection of distinct items.

Unordered means that items aren't stored in any particular order.

Something is either in the set or it is not in the set.

There is no notion of there being a first, second or last item.

Distinct means that any item appears in a set at most once, i.e., there are no duplicates.

We use curly brackets …

```
>>> vowels = {'a','e','i','o','u'}
>>> vowels
{'u', 'a', 'i', 'o', 'e'}
>>> vowels = {'a','e','a','a','i','o','u','u'}
>>> vowels
{'u', 'a', 'i', 'o', 'e'}
>>> type(vowels)
<class 'set'>
>>> type({1,2,3})
<class 'set'>
>>> set()
set()
>>> type(set())
<class 'set'>
>>> set([5,4,3,2])
{2, 3, 4, 5}
>>> set(5,4,3)
Traceback (most recent call last):
  File "<pyshell#90>", line 1, in <module>
    set(5,4,3)
TypeError: set expected at most 1 arguments, got 3
>>> set(range(5))
{0, 1, 2, 3, 4}
```

Set operations

In mathematics, set operations include union, intersection, add, and remove.

In Python, these are implemented as methods. There is a good description of these methods at: https://www.w3schools.com/python/python_ref_set.asp

```
>>> A = set(range(10))
>>> A
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
>>> B = {0,1,2,3,4}
>>> B
{0, 1, 2, 3, 4}
>>> C = {1,3,5,7,9}
>>> C
{1, 3, 5, 7, 9}
>>> B.add(9)
>>> B
{0, 1, 2, 3, 4, 9}
>>> B.difference(C)
{0, 2, 4}
>>> B
{0, 1, 2, 3, 4, 9}
>>> B.intersection(C)
{1, 3, 9}
>>> B.issubset(A)
True
>>> B.issuperset(C)
False
>>> B.remove(0)
>>> B
{1, 2, 3, 4, 9}
>>> B.union(C)
{1, 2, 3, 4, 5, 7, 9}
>>> C.union(B)
{1, 2, 3, 4, 5, 7, 9}
>>> B.clear()
>>> B
set()
```

Set contents must be immutable

The idea behind using sets is that checking for membership of a set must be fast.

A set is mutable, i.e., the elements in the set can be changed. For example we can use the add method.

But the elements of a set must be immutable, such as numbers and strings.

Therefore you cannot have a list as an element of a set and you cannot have a set as an element of a set.

3. Storing data using Tuples

Lists are not the only kind of ordered sequence in Python.

A string is an immutable sequence of characters and can be indexed and sliced like a list to create new strings.

```
>>> butterfly = 'red admiral'
>>> butterfly
'red admiral'
>>> butterfly[10]
'l'
>>> butterfly[0:3]
'red'
>>> butterfly[-5:]
'miral'
```

Python also has an immutable sequence type called a *tuple*.

Tuples are written using round brackets ( ).

Like strings and lists, tuples can be indexed, sliced, and looped over.

```
>>> greetings = ('hello','bonjour','hola','konnichiwa')
>>> for word in greetings:
      print(word)
```

```
hello
bonjour
hola
konnichiwa
```

Once a tuple is created, it cannot be mutated.

```
>>> greetings[0]
'hello'
>>> greetings[0] = 'hi'
Traceback (most recent call last):
  File "<pyshell#151>", line 1, in <module>
    greetings[0] = 'hi'
TypeError: 'tuple' object does not support item assignment
```

There is one small catch regarding the empty tuple.

```
>>> ()
()
>>> type(())
<class 'tuple'>
>>> (3+5)
8
>>> type((3+5))
<class 'int'>
>>> (3+5,)
(8,)
>>> type((3+5,))
<class 'tuple'>
```

4. Storing data using Dictionaries

A *dictionary* is an unordered mutable collection of key/value pairs.

They follow the idea of the usual dictionary of words that maps each word to its definition.

The keys form a set: any particular key can appear at most once in a dictionary.

Like the elements in a set, keys must be immutable.

Dictionaries are created by putting key/value pairs inside curly brackets. Each key is followed by a colon and then by its value.

```
>>> car = {'make':'Audi','model':'A3','year':2012}
>>> car
{'make': 'Audi', 'model': 'A3', 'year': 2012}
>>> car['make']
'Audi'
>>> car['model']
'A3'
>>> car['year']
2012
>>> for key in car:
      print(key,car[key])


make Audi
model A3
year 2012
```

We can also add key/value pairs to an existing dictionary.

```
>>> ship = {}
>>> type(ship)
<class 'dict'>
>>> ship['name'] = 'USS Enterprise'
>>> ship['reg'] = 'NCC-1701'
>>> ship['captain'] = 'James T. Kirk'
>>> ship['commissioned'] = 2245
>>> ship['propulsion'] = 'warp drive'
>>> ship
{'name': 'USS Enterprise', 'reg': 'NCC-1701', 'captain': 'James T. Kirk',
'commissioned': 2245, 'propulsion': 'warp drive'}
```

Dictionary operations

Like lists, tuples and sets, dictionaries are objects so they have methods:
https://www.w3schools.com/python/python_ref_dictionary.asp

```
>>> car.keys()
dict_keys(['make', 'model', 'year'])
>>> car.values()
dict_values(['Audi', 'A3', 2012])
>>> car.items()
dict_items([('make', 'Audi'), ('model', 'A3'), ('year', 2012)])
>>> list(car.keys())
['make', 'model', 'year']
>>> set(car.keys())
{'make', 'year', 'model'}
>>> list(car.values())
['Audi', 'A3', 2012]
>>> list(car.items())
[('make', 'Audi'), ('model', 'A3'), ('year', 2012)]
```

We can loop over the key/value pairs.

```
>>> for key in car:
      print(key,car[key])


make Audi
model A3
```

```
year 2012
>>> for key,value in car.items():
      print(key,value)


make Audi
model A3
year 2012
```

In a dictionary, the keys must be immutable. A tuple is immutable.

```
model A3
>>> coordinates = {(0,0):5, (0,1):3, (1,0):7, (1,1):2}
>>> coordinates
{(0, 0): 5, (0, 1): 3, (1, 0): 7, (1, 1): 2}
>>> set(coordinates.keys())
{(0, 1), (1, 0), (0, 0), (1, 1)}
```

5. Debugging

"If debugging is the process of removing software bugs, then programming must be the process of putting them in." — E.W. Dijkstra

We all write buggy code. Accept it.

Write you code with testing and debugging in mind.

Give your variables meaningful names.

Save a working copy (in case you subsequently mess it up).

Testing vs Debugging

Testing – Process of finding bugs (errors) in software. Done manually by a tester or can be automated.

Debugging – Process of fixing bugs found in the testing phase. The programmer is responsible for debugging and it can't be automated.

Using the Debugger in Spyder

```
import random
def rps():
    cd = 0
    cw = 0
    cl = 0
    for i in range(30):
        a = random.randint(0,3)
        b = random.randint(0,3)
        if (a==0):
            if (b==0):
                cd = cd+1
            elif (b==1):
                cl = cl+1
            else:
                cw = cw+1
        elif (a==1):
            if (b==0):
                cw = cw+1
            elif (b==1):
                cd = cd+1
            else:
                cl = cl+1
        else:
            if (b==0):
                cl = cl+1
            elif (b==1):
                cw = cw+1
            else:
                cd = cd+1
    return(cw,cd,cl)

print(rps())
```

If in doubt, add print statements.

Set a breakpoint near the start of a function using F12.

Step through the code one line at a time using Ctrl+F10.

Notice the values of the variable changing in the “Variable explorer” in the top right window pane.

Also notice the *control flow* through the code.

Summary

- Follow Python coding style to help yourself and others understand your code.
- Sets are used in Python to store unordered collections of unique values. They support the same operations as sets in mathematics.
- Tuples are another kind of Python sequence. Tuples are ordered sequences like lists, except they are immutable.
- Dictionaries are used to store unordered collections of key/value pairs. The keys must be immutable, but the values need not be.
- Looking things up in sets and dictionaries is much faster than searching through lists.
- Use a debugger to help with debugging code and assert statements to help with testing code.

Data structure	*Mutable?*	*Ordered?*	*When to use*
str	no	yes	You want to keep track of text.
list	yes	yes	You want to keep track of an ordered sequence that you want to update.
tuple	no	yes	You want to build an ordered sequence that you know won't change or that you want to use as a key in a dictionary or as a value in a set.
set	yes	no	You want to keep track of values, but order doesn't matter, and you don't want to keep duplicates. The values must be immutable.
dictionary	yes	no	You want to keep a mapping of keys to values. The keys must be immutable.