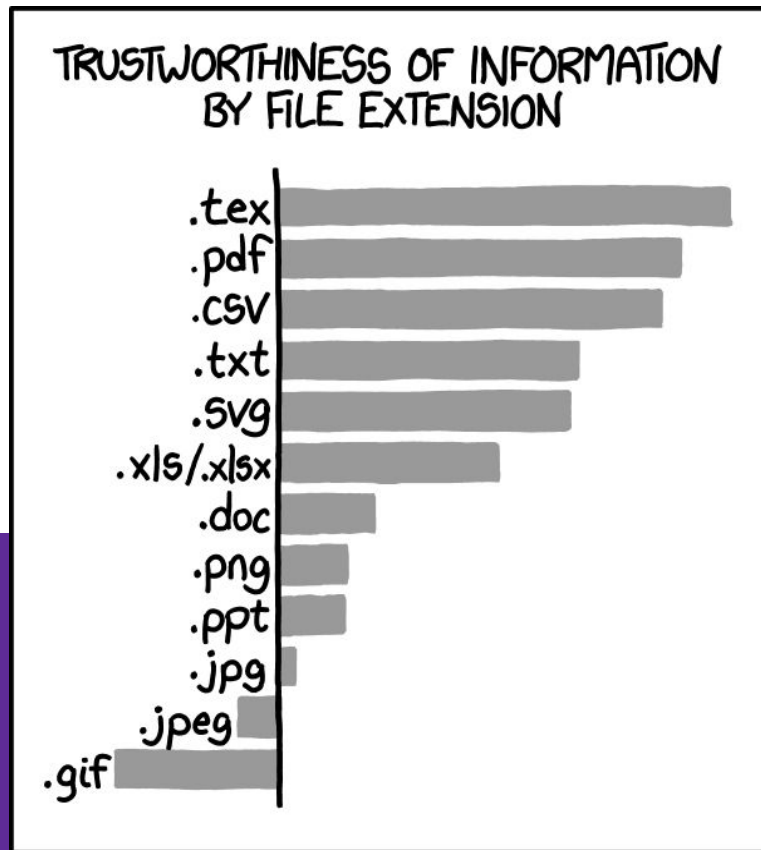


CS 1: Intro to CS

Intro to File Input/Output,
Dictionaries and Tuples



Source: <https://xkcd.com/1301/>

Administrivia

MP3 covers file processing, program decomposition with modules, and basic tuples/dictionaries

- Please start this one early!

Come to OH! No specific questions?

- Bring practice exercises from lecture, talk through your strategy with a TA
- Draw out each function for MP 3 on paper or as function stubs (what are the argument types? Any return? Edge cases to handle/document?)
- Get help on any VSCode debugging or pycodestyle questions
- Ask a TA about their experiences in CS/advice for CS 1 students
- Or, just come in and work on MP 3/HW 1 Rework. And say hi!

Learning Objectives

Know how to read and write files

Identify the difference between `readlines()` and `readline()` to process files

Identify common pitfalls in file processing:

- 1) Common (semantic) bugs
- 2) Program design

Preview dictionary/tuple syntax (continued Wednesday)

Practice various applications of file-processing, pitfalls, and program design (continued Wednesday)

Files

Data that programs can act on are either temporary or permanent

So far, all of our data has been temporary (e.g. variables and functions disappearing when the program is done)

We often want to work with permanent data though

Files

Data stored on computers are generally in the form of files (e.g. `.txt`, `.csv`, `.doc`, `.png`, [`.pdb`](#), `.py`, etc.)

Files are said to be “persistent”

- Still around after the program exits
- Still around after the computer shuts off

Files are very useful, and many real-world applications need to store some data

Some Example Applications with Files

Imagine you have a huge file which described the average size of raspberries in Finland for each day between from 1923 to 2011, and you want to find the lowest value. Then you can use python to read this file and find the lowest value.

Reading a .txt file of student names in a class to create a randomly generated seating chart.

You could take in a text file and see what the five most commonly used words were in order to generate tags to describe it.

Some Example Applications with Files

For machine learning applications, you may want to read in data from a CSV file to train and test your model on.

An input .csv file can contain the luminosity of a star over a period of time. It can be used as an input to a program that calculates the star's age using luminosity.

Tweets can be stored as a .txt file. a piece of python code reads the .txt file, and extracts all of the emojis from the tweets. so that the final output is a list of emojis.

A file could be used to store the statistics and personal attributes (e.g., number of wins in the game) of a person's character in a video game. This way, when the game is closed, it still has a way of accessing the permanent values that pertain to the character.

Some Example Applications with Files

Could load .txt files of books from different authors and look at the differences in the distributions of punctuation types across authors (just count every punctuation type).

We can read an image file, change the pixel data by applying a filter, and then display the new image.

Programming a 'choose-your-own-adventure' game, where the choices and storylines are labeled and defined in a .txt file

Some Example Applications with Files

Plagiarism checker: take in the file and finding the number of matches by comparing with other sources

Suppose you were working on a research project and some data that you downloaded off of a database came in the form of a .txt or .csv file. By reading this .txt file in your program (and converting the data from strings to the necessary type) you can complete the necessary computations to do your research!

Some Example Applications with Files

You can read the output of another program. For example, you can run fast code in C and read an output in python for post-processing.

Opening a set of astronomy stellar coordinates to find distances between stars.

Natural language processing! You need to read a file input and output a new file that processed what the inputted file said.

One application will be reading a .txt file with DNA sequences, search for and print out position of a specific sequence of interest, or compare two strings in different files for similarities and differences, calculate affinity to a protein according to features of the sequence.

Returning to our AQI Application

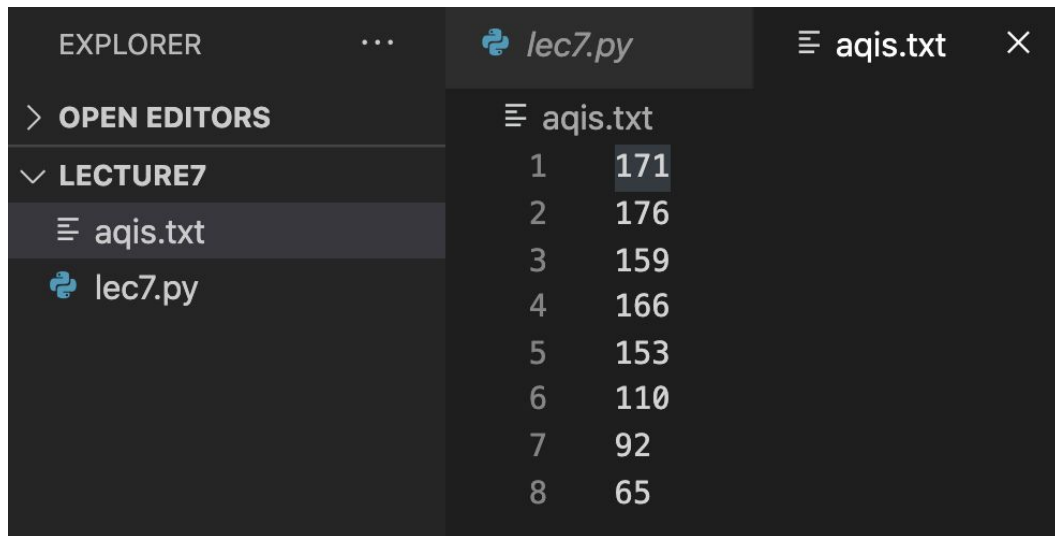
```
>>> aqis = [171, 176, 159, 166, 153, 110, 92, 65]
>>> aqis_above_150 = 0
>>> for value in aqis:
...     if value > 150:
...         aqis_above_150 += 1
...
>>> print('{} of {} days were unhealthy'.format(aqis_above_150,
...                                               len(aqis)))
...
5 of 8 days were unhealthy
```

Returning to our AQI Application

Recall our AQI program from Friday, counting the number of AQI values from Sept. 12th-19th 2020 that are above 150 (unhealthy to go outside)

We can change our program to store these values in a files (say, `aqis.txt`) instead of a temporary list variable

How do we get the values to use in our code?



The screenshot shows a code editor interface. On the left, the Explorer sidebar is visible, showing a folder named 'LECTURE7' containing two files: 'aqis.txt' and 'lec7.py'. The main editor area displays the content of 'aqis.txt', which is a list of eight integers: 171, 176, 159, 166, 153, 110, 92, and 65, each on a new line.

1	171
2	176
3	159
4	166
5	153
6	110
7	92
8	65

Opening a file

Files in Python are represented as **file objects**

These are not the same as the file, just a convenient way to interact with the file in Python

First, we need to create a file object in Python linked to the real file, then we can use [file methods](#) on that object

File objects are created using the built-in open function

Opening a file

`open(<name of file>, <mode>)`

- `<name of file>` is just the file's name as a string
- `<mode>` determines how you can interact with the file object

```
>>> f = open('aqis.txt', 'r')
>>> f
< io.TextIOWrapper name='aqis.txt' mode='r' encoding='UTF-8'>
```

Now the file `aqis.txt` has a corresponding Python object called `f` (a common variable name for file objects, but you can change it)

Method calls on `f` will do something to the file `aqis.txt`

Ways of opening a file

Three typical values of `<mode>`:

- `'r'` means that file has been opened “read-only” (you can't change the file, and it must already exist)
- `'w'` means that file has been opened for **w**riting (creating a new file from scratch, rewriting it if it already exists)
- `'a'` means that file has been opened for **a**ppending (file must exist, will be changed)

Since we're only reading `aqis.txt`, we use `'r'`

Closing a file

Once we're done working with a file object, we should close it (prevents further actions from occurring to the file)

```
f = open('aqis.txt', 'r')
# do stuff with file object f
f.close()
```

If we forget, the object will be closed when the program exists, but this is poor programming practice

We'll start with this approach to better understand the way files are processed, then show the preferred **with** syntax (which doesn't require an explicit `f.close()`) which we'll use moving forward (and which we encourage you to use in MP 3)

Reading from files

When reading from text files, can think of them as a bunch of lines (strings ending with the '`\n`' character)

Python methods for reading from text files:

- **readline()** - reading a **single** line from a text file (returns a string)
- **readlines()** - read **all** the lines of the text file (returns a list of strings)
- **read()** - read **all** the lines of the text file (returns a single string with lines separated by the '`\n`' character)

Note: The last two methods should rarely be used, as they are less efficient than reading line-by-line; keep this in mind in your labs/MPs!

f.readlines()

```
>>> f = open('aqis.txt', 'r')
>>> lines = f.readlines()
>>> lines
['171\n', '176\n', '159\n', '166\n', '153\n', '110\n', '92\n', '65']
```

Notes:

- Each element is a string, need to convert it to a number before using it
- Each string ends in a newline character (except the last)
- Changing the elements of the list **will not** change the contents of the file to change (the list and file are independent once `readlines` completes)

f.readline()

```
>>> f = open('aqis.txt', 'r')
>>> line = f.readline()
>>> line
'171\n'
>>> line = f.readline()
>>> line
'176\n'
```

Important notes:

- If there are no more lines (end of file) **readline** returns the empty string ''
- An empty line in the file will return a line which consists only of the newline character '\n'

What if you want to remove '`\n`'?

```
>>> f = open('aqis.txt', 'r')
>>> line = f.readline()
>>> line
'171\n'
>>> line = f.readline().rstrip()
>>> line
'176'
```

Strings have a method called `.rstrip()` to remove whitespace characters on the right end of the string (`.lstrip()` removes on the left, `.strip()` removes from both sides)

We won't need this for our AQI example (`int(line)` will ignore the '`\n`') but you will want to use this in many other file-processing applications.

f.readline() with AQIs

```
>>> f = open('aqis.txt', 'r')
>>> aqi = int(f.readline())
>>> if aqi > 150:
...     aqis_above_150 += 1
...
>>> aqi = int(f.readline())
>>> if aqi > 150:
...     aqis_above_150 += 1
... # etc.
```

Problem: How do we know when to stop?

f.readline() with AQIs

Let's use our new while loop:

```
>>> aqis_above_150 = 0
>>> f = open('aqis.txt', 'r')
>>> line = f.readline()
>>> while line != '':
...     if int(line) > 150:
...         aqis_above_150 += 1
...         line = f.readline()
...
>>> aqis_above_150
5
```

DRY again

We have repeated code again!

```
>>> aqis_above_150 = 0
>>> f = open('aqis.txt', 'r')
>>> line = f.readline()
>>> while line != '':
...     if int(line) > 150:
...         aqis_above_150 += 1
...         line = f.readline()
...
>>> aqis_above_150
5
```

A DRYer Solution

```
>>> aqis_above_150 = 0
>>> f = open('aqis.txt', 'r')
>>> while True:
...     line = f.readline()
...     if line == '':
...         break
...     if int(line) > 150:
...         aqis_above_150 += 1
...
>>> aqis_above_150
5
```


Using for with files

Python allows a useful shortcut using the **for** statement:

```
f = open('aqis.txt', 'r')
aqis_above_150 = 0
for line in f:
    if int(line) > 150:
        aqis_above_150 += 1
print('There were {} days above 150'.format(aqis_above_150))
f.close()
```

This is the preferred way to write this

The `with` Statement

So far, we've seen the following pattern for file processing:

```
f = open('aqis.txt', 'r')  
# use file...  
f.close()
```

It's very common to forget to close the opened file...

A common approach for file-processing in Python works with the `with` statement

The with Statement

We can instead write this:

```
with open('aqis.txt', 'r') as f:  
    # use file...
```

This will close the file automatically when the **with** block is exited (even if an exception is raised in the block)

**Do not use `f.close()`
if using `with`**

```
# f = open('aqis.txt', 'r')  
with open('aqis.txt', 'r') as f:  
    line = ''  
    while line != '':  
        line = f.readline()  
        if int(line) > 150:  
            aqis_above_150 += 1  
  
# f.close() # not needed using with
```

Writing Files

So far, we've seen how to read files. But what about the 'w' and 'a' permissions?

The complement of reading lines is writing lines:

```
with open('new_file.txt', 'w') as f:
    f.write('Hello world!\n')
    f.writelines(['Line 1', 'Line 2'])
    # f.close() by default using with
```

Be careful though, opening a file with 'w' will overwrite any existing file! Use 'a' to append lines to the end of an existing file.

Summary

Files are represented as file objects in Python

Files must be “opened” before use (creates a file object) and “closed” after use

- We can use the **with** statement as another way to open files to implicitly close them

The **readlines()** and **readline()** methods can be used to read data from a file line-by-line

- Be careful with the `\n` at the end of each line depending on your situation!

Wrap-Up: Preview of Tuples/Dictionaryes

(Code demo, continued Wednesday)

Case Study: Scantron Grader

Code Demo (starter files on [course website](#))

Summary of Monday's class exercise:

- Demoing `scantron_gen_test_files.py` as a way to automate sample-file-generation with file-writing
- Understanding the problem (side-by-side comparison of key and student Scantron file)
- Translating your approach to "grading by hand" into Python with file-processing, loops, and variables
- Implementing `grade_scantron(filename)` in `scantron_grader_starter.py`
- CYU: Why shouldn't we use `readlines()` or `read()`? How many loops did we end up needing to process both files together?

Next Time

More tuples/dictionaries

Tuples

Tuples are another Python data type

Basically, a “read-only” list that cannot be modified

Tuple syntax is simple:

(1, 2, 3, 4, 5)

Tuples of length 1 written like this: **(1,)**

Because **(1)** is just the number 1 (normal use of parentheses)

Empty tuple is: **()**

Tuples

Tuples support many, but not all, of the list operations

```
>>> nums_tup = (1, 2, 3)
>>> for n in nums_tup:
...     print(n)
...
1
2
3
>>> len(nums_tup)
3
>>> sum(nums_tup)
6
>>> nums_tup
(1, 2, 3)
```

Tuples

Similar to lists and strings, can concatenate tuples with the `+` operator and index with `[idx]`:

```
>>> (1, 2, 3) + ('foo', 'bar', 'baz')
(1, 2, 3, 'foo', 'bar', 'baz')
>>> str_tup = ('foo', 'bar', 'baz')
>>> str_tup[0]
'foo'
>>> str_tup[-1]
'baz'
```

Check your understanding: What does `str_tup[0][0]` evaluate to?

Tuples are Immutable!

Unlike a list and like a string, you **cannot change the contents of a tuple**:

```
>>> str_lst = ['foo', 'bar', 'baz']
>>> str_tup = ('foo', 'bar', 'baz')
>>> str_lst[0] = 'hello'
>>> str_lst
['hello', 'bar', 'baz']
>>> str_tup[0] = 'hello'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Why tuples?

Tuples are basically a more restricted kind of list. So what good are tuples, anyway?

Not really an essential language feature

- Python could do fine without them, and many languages don't support them

There are some cases where tuples are convenient:

1. Returning multiple values from functions
2. Tuple unpacking
3. for loops with multiple bindings

1. Multiple return Values

Functions in Python can only return a single value (most of the time, this is all we need)

Sometimes, it's useful to be able to return more than one value from a function (e.g. counts of each vowel type in a given string)

We do this by creating a tuple of the returned values, and returning that one value

Example

One built-in function is `divmod`, which returns both the quotient and remainder of its arguments as a tuple

```
>>> divmod(10, 3)
(3, 1)
>>> divmod(42, 7)
(6, 0)
>>> divmod(101, 5)
(20, 1)
>>> divmod(-10, 3)
(-4, 2)
```

Example

We could define `divmod` ourselves:

```
>>> def divmod(m, n):  
...     return (m // n, m % n)  
...  
>>> qr = divmod(101, 5)  
>>> quotient = qr[0]  
>>> remainder = qr[1]  
>>> quotient  
20  
>>> remainder  
1
```


2. Tuple Unpacking

We can use `divmod` even more elegantly with what's known as **tuple unpacking**:

```
>>> qr = divmod(101, 5)
>>> (quotient, remainder) = qr
>>> quotient
20
>>> remainder
1
>>> qr
(20, 1)
```

Python also lets us write this without parentheses:

```
>>> quotient, remainder = qr
>>> quotient, remainder = divmod(101, 5)
```

Tuple Unpacking

Structure of a tuple unpacking:

```
(a, b, c) = t
```

`t[0]` is assigned to `a`, `t[1]` is assigned to `b`, `t[2]` is assigned to `c`

Tuple `t` must already be defined as a tuple and have the same number of elements on left-hand side, otherwise an error will occur

```
>>> tup = (1, 2, 3)
>>> a, b, c = tup
>>> a, b, c, d = tup
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: not enough values to unpack (expected 4, got 3)
```

Using tuples to swap Two Variables

Tuple unpacking lets us do a cute trick: swapping the value of two variables

```
>>> a = 10
>>> b = 20
>>> (a, b) = (b, a)
>>> a
20
>>> b
10
```

Why does this work?

3. for Loops with Multiple Bindings

We can also use tuples with for loops to assign (“bind”) values to multiple names every time through the loop:

```
>>> for (n, s) in [(1, 'a'), (2, 'b'), (3, 'c')]:  
...     print('num = {}, char = {}'.format(n, s))  
...  
num = 1, char = a  
num = 2, char = b  
num = 3, char = c
```

We'll get back to this...

Check Your Understanding

What are two differences between lists and tuples?

What are one similarities between tuples and strings?

Identify an example of a variable or function return that makes sense as:

1. A **tuple** instead of a **list** or **str**
2. A **list** instead of a **str** or **tuple**
3. A **str** instead of a **list** or **tuple**

Returning to the enumerate Function

Last week, we introduced `range` with the following `for` loop example:

```
>>> nums = [23, 12, 45, 68, -101]
>>> for i in range(len(nums)):
...     nums[i] *= 2
...
>>> nums
[46, 24, 90, 136, -202]
```

The purpose of `range(len(nums))` is to produce a list of the indices of `nums`

Seems like a lot of work for something so simple...

The enumerate Function

In Lab02, we introduced you to the `enumerate` function to write code like this:

```
>>> nums = [23, 12, 45, 68, -101]
>>> for (i, e) in enumerate(nums):
...     nums[i] *= 2
...
>>> nums
[46, 24, 90, 136, -202]
```

`enumerate` takes a sequence and outputs the indices (**i**) and elements (**e**) of the sequence one by one; we can now identify this (**i**, **e**) as a tuple!

The enumerate Function

Conceptually, it's as if **enumerate** produces a list of (index, element) tuples:

```
>>> enumerate(['a', 'b', 'c'])  
[(0, 'a'), (1, 'b'), (2, 'c')]
```

This is not actually what happens...

```
>>> enumerate(['a', 'b', 'c'])  
<enumerate object at 0x10f7d0f40>
```

But it *is* how it behaves inside a **for** loop. Like with **range**, we can see this more clearly with the **list()** wrapper:

```
>>> list(enumerate(['a', 'b', 'c']))  
[(0, 'a'), (1, 'b'), (2, 'c')]
```


The enumerate Function

enumerate objects, like **range** objects, contain an **iterator** that allows them to be used inside a **for** loop

Lots of Python objects have associated iterators!

- lists, strings, tuples (sequences)
- files
- **range** objects
- **enumerate** objects
- dictionaries (we'll see these later)
- ... and you can define your own (we'll also see this later)

The enumerate Function

Back to our example:

```
>>> nums = [23, 12, 45, 68, -101]
>>> for (i, e) in enumerate(nums):
...     nums[i] *= 2
...
>>> nums
[46, 24, 90, 136, -202]
```

Note that we don't need the `e` variable anywhere in this example

Can name `e` anything we want (doesn't matter)

- But cannot leave it out, or `i` will become a tuple
- Usually we name it `_` (indicates unused name)

The enumerate Function

What happens when we leave e out?

```
>>> nums = [23, 12, 45, 68, -101]
>>> for i in enumerate(nums):
...     print(i)
...
(0, 23)
(1, 12)
(2, 45)
(3, 68)
(4, -101)
```

Instead, can do:

```
>>> nums = [23, 12, 45, 68, -101]
>>> for (_, i) in enumerate(nums):
...     print(i)
...
23
12
45
68
-101
```

The enumerate Function

Compare:

```
for i in range(len(nums)):
```

with:

```
for i, e in enumerate(nums):
```

Which one you use is up to you...

- Both are acceptable, but **enumerate** is considered to be better style

Dictionaries

A **dictionary** is a new kind of Python data type

Before we describe what it is, let's describe a problem it could solve...

Caltech Usernames

Each member of Caltech has a unique username (e.g. *hovik* for El Hovik, *gmccabe* for Gavin McCabe)

These usernames are used in various contexts, such as constructing a Caltech email (*hovik@caltech.edu*, *gmccabe@caltech.edu*)

How can we easily keep track of which username is associated with which full name?

Caltech Usernames

For each individual, we need:

- The full name of the individual
- Their unique username

Given what we know now, how could we do this?

Lists?

You could have a list of names and usernames:

```
usernames = ['El Hovik', 'hovik',  
            'Gavin McCabe', 'gmccabe', ...]
```

But it would not be easy to find the username corresponding to a different name

It would be better if a name and the corresponding username were connected in some way

List of tuples?

You could have a list of (name, username) tuples:

```
usernames = [('El Hovik', 'hovik'),  
             ('Gavin McCabe', 'gmccabe'), ...]
```

Let's see what we would need to do in order to find the username corresponding to a particular name (e.g. "Gavin McCabe")

List of tuples?

We could write code like this:

```
>>> for name, username in usernames:
...     if name == 'Gavin McCabe':
...         print('Username for {}: {}'.format(name, username))
...
Username for Gavin McCabe: gmccabe
```

This is not bad, but:

- Can't modify username (tuples are immutable)
- Have to look through entire list in worst case to find one username
- Cumbersome!

Back to Dictionaries

A dictionary is a data structure that stores associations between **keys** and **values**

In the previous example:

- **key**: the individual's name
- **value**: the username

Dictionaries make it easy to:

- Find the value given the key
- Change the value given the key
- Add more key/value associations

And they're efficient!

Keys and Values

The **values** stored in a dictionary can be any Python value

Keys can only be *immutable* (unchangeable) Python values, e.g.:

- strings
- tuples
- numbers (rare)

Usually, we use strings as keys

Some Examples of Dictionaries

Student ideas:

“If you were programming a game of Scrabble, you would need to associate a point value with each letter of the alphabet.”

“For a chemical reaction, we could have molecules (strings) map to their concentrations taken at many intervals (could be string of number values, or a list or tuple).”

“Year mapped to number of potatoes grown in Ireland for that year.”

“An item in a store (string keys) associates to a price (float values).”

“You could map Lego set ID numbers to the names of the actual sets themselves. For example, 75280 would be matched to 501st Legion Clone Troopers.”

Some Examples of Dictionaries

“One example is the texting shortcuts on iPhones, where it will have a list of abbreviations to automatically change to longer phrases.”

*“The Dewey Decimal System for a library is an example of this. In this case, the numbers 000-999 are used as **keys** to tell the readers what subject their book will be about - the **values** (e.g. 920 --> biographies; 400 --> languages)”*

“Journal articles on the internet map to specific DOI (digital object identifier) values, which people use to find the exact journal article they are looking for.”

Some personal favorites used for web data... [Reddit posts](#) and [Pokemon](#)

Dictionary Syntax

The contents of a dictionary are written between curly braces (`{` and `}`)

The empty dictionary (no key/value pairs) is written as: `{}`

In general, a dictionary is a comma-separated collection of **key** : **value** pairs:

```
{ key1 : value1, key2 : value2, ... keyN : valueN }
```

An example dictionary might look like this:

```
{ 'El Hovik' : 'hovik', 'Gavin McCabe' : 'gmccabe' }
```

Getting a value given a key

Suppose we want to get Gavin's username from our (mini) dictionary of usernames:

```
>>> usernames = {'El Hovik' : 'hovik',  
...              'Gavin McCabe' : 'gmccabe'}  
>>> usernames['Gavin McCabe']  
'gmccabe'
```

Note that this looks like accessing a *list* with a value of **'Gavin McCabe'**

Python is “overloading” the meaning of the square brackets

With list indexing, the value inside brackets **can only be an integer** (the index of the list)

With a dictionary, it's **any key value**

Changing a value at a key

Suppose Gavin wants to change his username with an alias.

Can change the dictionary value too:

```
username['Gavin McCabe'] = 'gavin'
```

Like the syntax for changing a list value, except that the “index” is a string, not a number

Adding a new key/value pair

We add a new key/value pair to a dictionary by “changing” a key that wasn’t there before:

```
>>> usernames = {'El Hovik' : 'hovik', 'Gavin McCabe' : 'gmccabe'}
>>> usernames['Mike Vanier'] = 'mvanier'
>>> usernames['Adam Blank'] = 'blank'
>>> usernames
{'El Hovik': 'hovik', 'Gavin McCabe': 'gmccabe', 'Mike Vanier': 'mvanier',
 'Adam Blank': 'blank'}
```

Accessing a nonexistent key

Here's what happens if you try to access a key that isn't in the dictionary:

```
>>> usernames['Curly Vanier']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Curly Vanier'
```

Deleting a key/value pair

We use `del` to remove a key/value pair from a dictionary:

```
>>> del usernames['Adam Blank']
>>> usernames
{'El Hovik': 'hovik', 'Gavin McCabe': 'gmccabe', 'Mike Vanier': 'mvanier'}
>>>
```

`del` is actually a special Python statement

- It's not a function, so no parentheses around its argument

`del` can remove elements from things other than dictionaries (e.g. lists)

- But more useful with dictionaries than lists

Back to our example

Let's improve the example by using a tuple of first and last names as keys:

```
>>> usernames = {('El', 'Hovik') : 'hovik',  
...               ('Gavin', 'McCabe') : 'gmccabe',  
...               ('Mike', 'Vanier') : 'mvanier'}
```

Does this work?

- This is ok because both tuples and strings are immutable (remember that keys must be immutable)

Back to our example

Can access **usernames** using **tuple** as key (but not single tuple components):

```
>>> usernames[('El', 'Hovik')]
'hovik'
>>> usernames['El']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'El'
>>> usernames['Hovik']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Hovik'
```

Dictionaries and for loops

We've seen many things that can be looped over using **for** loops:

- lists
- strings
- files

We can also use a **for** loop over dictionaries

```
for key in usernames:  
    print(key)
```

Looping over a dictionary loops over the *keys* in the dictionary (not the values)

Dictionaries and for loops

Looping over the keys in our dictionary of usernames:

```
>>> for key in usernames:  
...     print(key)  
...  
( 'El', 'Hovik' )  
( 'Gavin', 'McCabe' )  
( 'Mike', 'Vanier' )
```


Dictionaries and for loops

However, we usually want the values, not the keys: How can we modify our loop to print the values instead?

```
>>> for key in usernames:  
...     print(usernames[key])  
...  
hovik  
gmccabe  
mvanier
```

More on Booleans

Booleans are **True/False** values

However, Python has a notion of “truthiness” for non-boolean values

Several things besides the **False** value are considered to be **False** by Python:

- The empty string: ''
- The empty list: []
- The number 0

Most other values can be considered **True**

Boolean Operators

Python also supports three built-in boolean operators: **not**, **and**, and **or** (all take boolean arguments and return booleans)

- **not** - negates the boolean value

```
>>> not True
False
>>> not False
True
>>> not 0
True
>>> not 1
False
>>> not ''
True
>>> not []
True
```

Boolean Operators

and and **or** are operators that take two boolean arguments and return a boolean value

- **and** returns **True** if both values are **True**, otherwise **False**
- **or** returns **True** if either value is **True**, otherwise **False**

```
>>> True and True
True
>>> True and False
False
>>> False or True
True
>>> False or False
False
```

Boolean Operators

Generally use **and** and **or** with expressions that *evaluate* to boolean values (e.g. relational operators)

```
a = 10
b = 30
# ... later in code ...
if a > 0 and b < 50: ...
# ... later in code ...
if a < 0 or b > 50: ...
```

The `in` operator

Last time, we saw the `in` operator in sequences:

```
>>> 1 in [1, 2, 3]
True
>>> 'b' in 'foobar'
True
```

Can also use `in` with dictionaries, where `<key> in <dictionary>` means: is the **key** `<key>` one of the keys in `<dictionary>`?

```
>>> 'foo' in {'foo' : 1, 'bar' : 2}
True
>>> 1 in {'foo' : 1, 'bar' : 2}
False
```