Week 10 - Wednesday

COMP 1800

Last time

- What did we talk about last time?
- Regular expressions

Questions?

Assignment 7

Review

Exam 2

- Format:
  - Multiple choice questions (~20%)
  - Short answer questions (~20%)
  - Programming problems (~60%)
- Written in class
  - No notes
  - Closed book
  - No calculator

Files

Files

- A file is a series of bytes stored on a computer
- Usually, a file is stored on a hard drive or SSD
- It's **persistent**, so it exists after a program is done running
- Files allow us to do input that would be tedious by hand
- Files also allow us to do output that is too long to read in one go

Opening a file

- We can open a text file with the **`open()`** function
- It takes two string arguments:
  - File name
  - Mode (reading: **`'r'`**, writing: **`'w'`**, or append: **`'a'`**)
- Append is like writing, except that append writes to the end of the file while writing destroys whatever used to be in the file

```python
file = open('data.txt', 'r')
```

Closing a file

- After you open a file and read from it or write to it, you need to close it
- Files take up resources on the system, so having too many open files is wasteful
- There can be issues with reading or writing a file that another program has open
- Some of your data might get lost if you're writing to a file and forget to close it before your program ends
- To close a file, call the file reference's **`close()`** method

```
file.close()
```

Using with/as

- Because it's annoying to have to remember to close a file, Python has syntax that makes it unnecessary
- This alternative style starts with the keyword **with**
- Then, code using the file is in an indented block

```
with open('data.txt', 'r') as file:
  # Do the reading you want to do with file
  # Do some calculations
```

- The file is automatically closed after the indented block

Using split() with files

- Each line of a file might contain several data fields.
- The **split()** method can be used to break a line into a list of fields
- For example, a comma-separated-value (CSV) file divides values with commas

```
with open('data.csv', 'r') as data:
  for line in data:
    for column in line.split(','):
        print(column)
```

File methods

- Here are a few useful file methods that can be used for reading or writing individual lines or characters:

`read()`	Reads entire file as a single string
`read(n)`	Reads **n** characters from file as a string
`readline()`	Reads the next line of the file
`readline(n)`	Reads **n** characters from the next line of the file
`readlines()`	Reads all the lines of the file as a list of strings
`readlines(n)`	Reads **n** lines of the file as a list of strings
`write(s)`	Write the string **s** to the file

- Each of these file methods would be called on an open file reference:

```
with open('data.txt', 'r') as data:
  firstLine = data.readline()
```

while Loops

Anatomy of a `while` loop

```
while condition :

    statement1
    statement2
    …
    statementn
```

A whole bunch of statements

Rules for `while`

- The **`while`** loop executes each statement one by one
- When execution gets to the bottom, it jumps to the top
- If the condition is still **`True`** (i.e., **`i < 100`**), it repeats the loop
- In Python, some tasks can only be done with a **`while`** loop because we don't know how many times they will repeat

List Comprehensions

A list comprehension for 10 perfect squares

- Code we already know using **`append()`**:

```
values = []
for i in range(10):
  values.append(i**2)
```

- List comprehension version:

```
values = [i**2 for i in range(10)]
```

A list comprehension for perfect squares of odd numbers

- Code we already know using **append()**:

```
values = []
for i in range(10):
  if i % 2 == 1:
    values.append(i**2)
```

- List comprehension version:

```
values = [i**2 for i in range(10) if i % 2 == 1]
```

List comprehension syntax

- A list comprehension looks like:

```
[expression for i in iterable if condition]
```

- The **`expression`** part is any single Python expression that generates a value (and usually involves your iterating variable)
- You can use any variable, **`i`** here is just an example
- The **`iterable`** is anything a **`for`** loop can loop over, like a string, another list, or a **`range()`** function
- The **`if condition`** part is optional

Reading Data from the Internet

URL

- **URL** is an abbreviation for Uniform Resource Locator
- Format: `protocol host resource parameters`
  - `http://faculty.otterbein.edu/wittman1/comp1800/`
  - `https://www.youtube.com/watch?v=GQf25_9NOts`
- Hosts are often given as domains
  - Top-level domain: **`edu`**
  - Second-level domain: **`otterbein`**
  - Subdomain: **`faculty`**

JSON (JavaScript Object Notation)

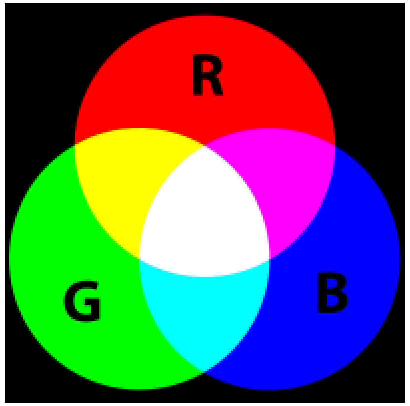
- JSON is an industry standard data structure for transmitting data across network connections
- It uses dictionaries and lists to create hierarchical and structured repositories of data that can be accessed programmatically
- JSON data itself is always a string
- Example JSON data:

```
'{"artist":"Led Zeppelin", "name":"Stairway to Heaven",
"length":"7:55", "year":1971}'
```

Images

RGB

- One system for representing color is **RGB**
- With **Red**, **Green**, and **Blue** components, you can combine them to make most visible colors
- Combining colors is an additive process:
  - With no colors, the background is black
  - Adding colors never makes a darker color
  - Pure **Red** added to pure **Green** added to pure **Blue** makes **White**
- **RGB** is a good model for computer screens



Pixels

- All computer images are made up of **pixels**
  - Short for **picture elements**
- Each pixel is a single color
- The smaller the pixels, the more realistic the image

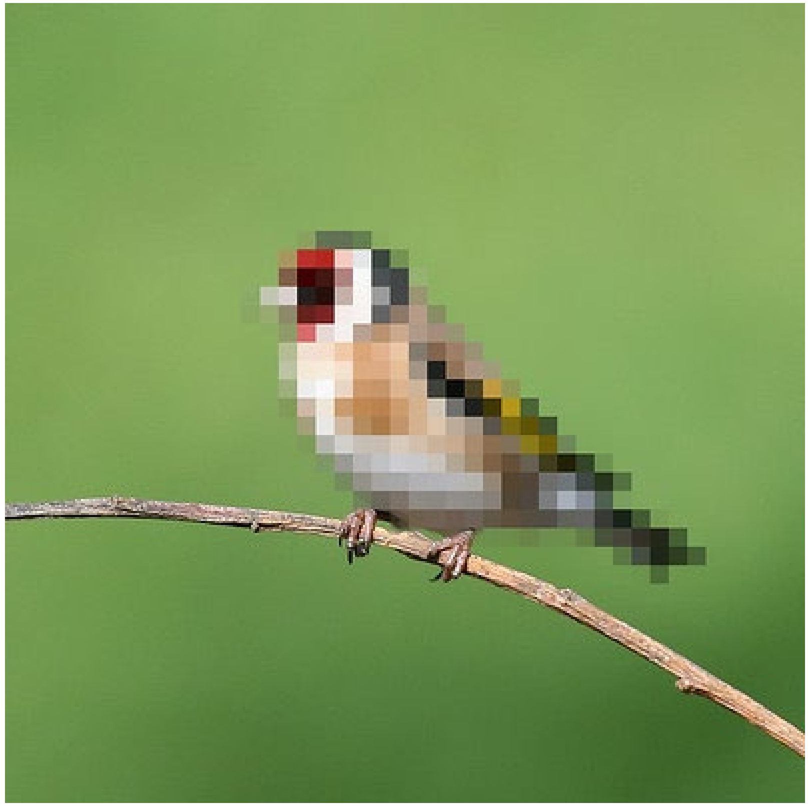


Image by Rego Korosi
https://www.flickr.com/photos/korosirego/4592913123/

To use Pixel

- To create a custom color:

```
color = Pixel(255,165,0) # orange
green = color.getGreen()
```

- Create colors using **Pixel** to specify RGB values
- Get individual values using:
  - getRed()
  - getGreen()
  - getBlue()

Image methods

Method	Use
`FileImage(file)`	Creates an **`Image`** object from a file name
`EmptyImage(width, height)`	Creates a blank **`Image`** of size **`width`** by **`height`**
`getWidth()`	Return the width of the image
`getHeight()`	Return the height of the image
`getPixel(x, y)`	Return the **`Pixel`** which is the color at (**`x`**,**`y`**)
`setPixel(x, y, pixel)`	Set the **`Pixel`** object at (**`x`**,**`y`**) to **`pixel`**
`save(file)`	Save the **`Image`** to the file with the given file name

Nested loops

- We can put loops inside of other loops
- Doing so is useful when we want to perform a repeated task as part of another repeated task
- Example:
    - Loop over every column in an image
        - For each column, loop over every row
- Code:

```
for x in range(picture.getWidth()):
    for y in range(picture.getHeight()):
        # do something
```

Namespaces

Builtins

- Some special functions are always available and don't need to be imported
- These are called **builtins**:

```
chr()
float()
input()
int()
len()
max()
min()
ord()
print()
range()
round()
str()
sum()
```

- IDLE shows these in **purple** font
- There are more, but these are the ones we've talked about in class

Importing a module

- Most of the imports in this class have been importing a module
- Doing so gives you access to code in the module
- But it also requires you to type the name of the module with using stuff from it

```
import math
print(math.pi)
print(math.sqrt(5))
```

Importing from a module

- If you don't want to type the name of a module, you can import functions or objects from the module

```
from math import pi
print(pi) # no math. needed!
```

- You can even import everything from a module, using the wildcard **`*`**

```
from math import *
print(pi) # math. is never needed again!
print(sqrt(5))
```

- The problem is that you will run into problems if something is named **`pi`** or **`sqrt`** in another module you import everything from

Function Variables

Putting a function in a variable

- What if what we wanted to store wasn't a value but was an **action** instead?
- We can store **functions** into variables
- All you have to do is use the name of the function without the parentheses

```
import math

action = math.sqrt # no parentheses, just the name
print(math.sqrt(5)) # prints square root of 5
print(action(5)) # also prints square root of 5
```

We can make a function that does anything

- This function will apply any function (called **`action`**) to everything in the list, with a given starting value

```
def process(values, action, starting):
    result = starting
    for value in values:
        result = action(result, value)
    return result
```

Let's make a few actions

- These functions are functions we can use with process
- One adds two numbers, and the other multiplies them

```
def add(a, b):
    return a + b

def multiply(a, b):
    return a * b
```

Using our actions

- Now we can call **`process`** with the actions we defined

```
numbers = [3, 4, 9, 2, 1, 7]
total = process(numbers, add, 0) # starts at 0
product = process(numbers, multiply, 1) # starts at 1
```

- We can even use a built-in function like **`max`**

```
largest = process(numbers, max, numbers[0])
```

Cryptanalysis

Cryptography

- "Secret writing"
- The art of encoding a message so that its meaning is hidden
- **Cryptanalysis** is breaking those codes

Encryption and decryption

- **Encryption** is the process of taking a message and encoding it
- **Decryption** is the process of decoding the code back into a message
- A **plaintext** is a message before encryption
- A **ciphertext** is the message in encrypted form
- A **key** is an extra piece of information used in the encryption process

Transposition cipher

- In a transposition cipher, the letters are reordered but their values are not changed
- Any transposition cipher is a permutation function of some kind

Brute force cryptanalysis

- **Brute force** means trying all possibilities
- For some kinds of encryption, that would mean trying trillions of possibilities
- For a rail fence cipher, the possible numbers of rails go from 2 up to the length of the message
- Thus, we can make a simple brute force function that runs our decryption algorithm with all possible rail sizes

```
def railBrute(ciphertext):
    for i in range(2, len(ciphertext) + 1):
        print(railDecrypt(ciphertext, i))
```

Automated brute force

- Although the previous function gets the right answer, we have to look at all the encryptions to see which one makes sense
- However, if we load a file containing English words into a Python dictionary, we could see how many real words show up in each decryption
- Then, we could store the one with the most real English words, assuming that is the best decryption

Simple monoalphabetic substitution cipher

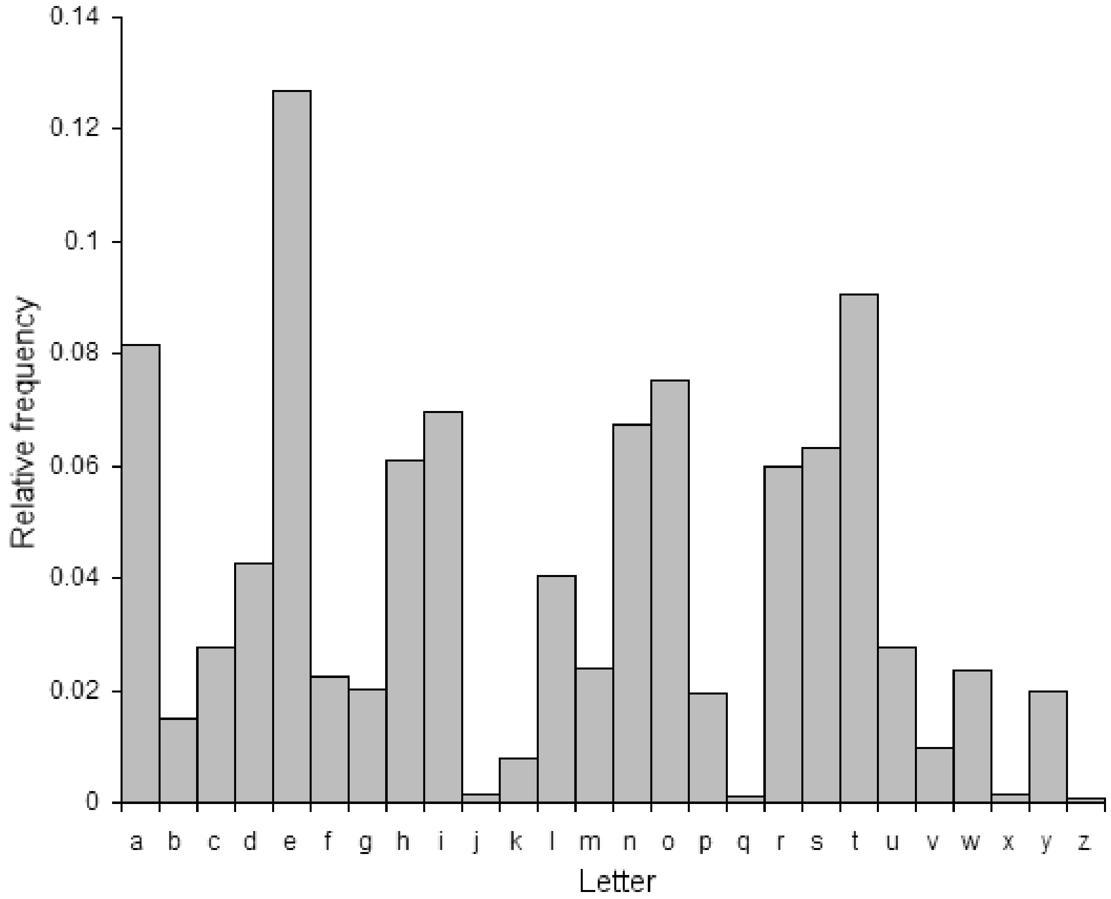
- We can map to a random permutation of letters
- For example:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
I	N	O	V	Z	H	A	P	T	R	G	E	U	F	D	W	S	B	Q	Y	L	K	M	J	C	X

- E("MATH IS GREAT") = "UIYP TQ ABZIY"
- 26! possible permutations
- Hard to check every one

Frequency attack

- English language defeats us
- Some letters are used more frequently than others: ETAOINSHRDLU
- Longer texts will behave more consistently
- Make a histogram, break the cipher



Tuples

- **Tuples** in Python are like lists, except that you can't change them
- You can still access the items in them with square brackets and an index number
- Instead of using square brackets `[]` to say what's in a tuple, you use parentheses `()`

```
things = (4, 'wombat', 2.9)
print(things[0]) # prints 4
print(things[1]) # prints wombat
print(things[2]) # prints 2.9
```

Sorting a list in an arbitrary way

- If you have a list (called, say, **`things`**), you can sort it with the sort function:

```
things.sort()
```

- But that only works if the items in things are items that Python knows how to sort, like strings or numbers
- If you want to sort arbitrary items, you have to pass in a function that says how you want them sorted, using a special named argument called **`key`**

```
things.sort(key=howToSort)
```

Sorting tuples

- In our case, we have a list of tuples that look like this:

```
('A', 0.08162203832186278)
```

- We want to sort by the second thing, the frequency
- We can write a simple function that gives the second thing (which has index 1) in a tuple

```
def second(pair):
    return pair[1]
```

Regular Expressions

What if you wanted to do partial matches with text?

- Maybe you want to search for text that:
  - Ends with "tion"
  - Starts with either "Password:" or "password:"
  - Has exactly five digits, like a zip code
  - Has a number followed by words like "street", "road", "avenue", "boulevard", "court", "way", or a few other possibilities
- The tool you want is called **regular expressions**
- Regular expressions can also be used to verify the formatting of data entered into websites

Regular expression syntax

- In Python, regular expressions are written as strings, using symbols that have special meanings

Symbols	Meaning	Example	Explanation
[]	Set of characters	'[m-z]'	Letters m through z
\	Special sequence	'\d'	Numerical digits
.	Any character (except newline)	'cr.p'	'crap', 'crip', 'cr8p', etc.
^	Starts with	'^the'	Line starts with 'the'
$	Ends with	'dog$'	Line ends with 'dog'
*	Zero or more occurrences	'hi*'	'h', 'hi', 'hii', 'hiii', etc.
+	One or more occurrences	'hi+'	'hi', 'hii', 'hiii', etc.
?	Zero or one occurrences	'team?'	'tea' or 'team'
{}	The specified occurrences	'he.{2}o'	'hello', 'helpo', 'hemno', etc.
|	Either/or	'gray|grey'	'gray' or 'grey'

Special sequences

- Because there are certain sets of characters used a lot, there are special sequences for those

Sequence	Meaning
`\d`	Numerical digit (0-9)
`\D`	**Not** a numerical digit
`\s`	White space (space, tab, etc.)
`\S`	**Not** white space
`\w`	Alphanumeric (A-Z, a-z, 0-9, and underscore)
`\W`	**Not** alphanumeric

Set syntax

- Sets of characters are used a lot
- There are special rules inside the brackets

Set Example	Meaning
`[amp]`	Either a, m, or p
`[a-n]`	Any lowercase character in the range from a to n
`[^amp]`	Any character except a, m, or p
`[0-9]`	Any digit 0-9
`[a-zA-Z]`	Any lowercase or uppercase letter
`[+]`	The character +, since most special characters have no special meaning **inside** sets

Raw strings

- Both regular expressions and Python strings use backslash (\) to mean special things
- For this reason, it's common to use **raw strings** in Python when specifying a regular expression
- Raw strings start with **r** (before the quotes) and don't treat backslashes as special characters
- Raw strings are still normal strings, they just let you type things in differently

```
word1 = '\n'  # contains newline
word2 = '\\n' # contains \n (two characters)
word3 = r'\n' # contains \n (two characters)
```

Python functions for regular expressions

- Once you have a string that represents a regular expression, how can you use it?
- First, import **`re`**
- The **`re`** module has a number of functions, but three will be useful for us:

Function	Description
`findall()`	Return a list of all the strings that match
`split()`	Split a string into a list separated by places that match
`sub()`	Replace matches with a string

Regular expression examples

```
import re

text = 'we are the wombat combat warriors'
# get all words that start with w
wWords = re.findall(r'w[a-z]*', text)
# Gets: ['we', 'wombat', 'warriors']

# split up the string by words that start with w
noWWords = re.split(r'w[a-z]*', text)
# Gets: ['', ' are the ', ' combat ', '']

# replace every word that starts with w with goat
newText = re.sub(r'w[a-z]*', 'goat', text)
# Gets: 'goat are the goat combat goat'
```

Studying Advice

Studying advice

- Focus on quizzes
- Focus on assignments
- Memorizing things about Python is okay
- Practicing programming is better

- Hints:
  - You will probably have to use dictionaries
  - You will have to do something with nested `for` loops, probably image manipulation
  - You will have to use `while` loops
  - You might have to explain a regular expression

Upcoming

Next time…

- Work day for Assignment 7

Reminders

- Review chapters 5 through 8 of the textbook
  - **Exam 2 on Monday!**
- Work on Assignment 7
  - **Due Friday by midnight!**