CS 115, Autumn 2021

Lecture 39: goodbye, world

Searching for additional programming topics

- Programmers generally do not memorize a lot of syntax
  - Instead they look up functions when they need them
  - Python official documentation: https://www.python.org/doc/

More Advanced Python

- We learned a lot this quarter but there is way more you can do with Python.

- Some topics you may want to explore:
  - tuples
  - dictionaries
  - lists that change size
  - lists of lists
  - list comprehensions
  - classes and objects

Tuples

- A sequence similar to a `list` but it **cannot be altered**
- Good for storing related data
  - We mainly store the same **type** of data in a `list`
  - We usually store related things in tuples

- Creating tuples

name = (**data**, **other_data**, … , **last_data**)

```
tuple = ("Tucson", 80, 90)
```

Using tuples

- You can access elements using [] notation, just like lists and strings

```
tuple = ("Tucson", 80, 90)
low = tuple[1]
```

- You cannot update a tuple!
  - Tuples are immutable
- You can loop through tuples the same as lists

operation	call	result
len()	`len((1, 2, 3))`	`3`
+	`(1, 2, 3) + (4, 5, 6)`	`(1, 2, 3, 4, 5, 6)`
*****	`('Hi!',) * 4`	`('Hi!', 'Hi!', 'Hi!', 'Hi!')`
in	`3 in (1, 2, 3)`	`True`
for	`for x in (1,2,3): print x,`	`1 2 3`
min()	`min((1, 3))`	`1`
max()	`max((1, 3))`	`3`

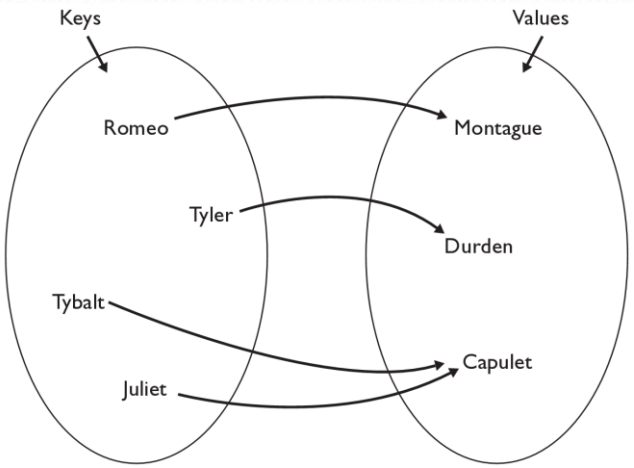
Exercise

- Write a program to count the number of occurrences of each unique word in a large text file (e.g. *Moby Dick*).
  - Allow the user to type a word and report how many times that word appeared in the book.
  - Report all words that appeared in the book at least 500 times.
- What structure is appropriate for this problem?

Dictionaries

- **dictionary**: Holds a set of unique *keys* and a collection of *values*, where each key is associated with one value.
  - a.k.a. "map", "associative array", "hash"

- basic dictionary operations:
  - **put**(*key*, *value*): Adds a mapping from a key to a value.
  - **get**(*key*): Retrieves the value mapped to the key.
  - **remove**(*key*): Removes the given key and its mapped value.



`my_dict["Juliet"]` returns `"Capulet"`

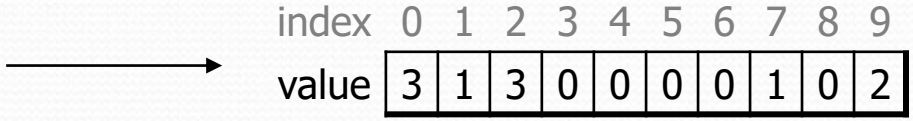
`Dictionary` functions

`my_dict[`**`key`**`] = `**`value`**	adds a mapping from the given key to the given value; if the key already exists, replaces its value with the given one
`my_dict[`**`key`**`]`	returns the value mapped to the given key (error if key not found)
`items()`	return a new view of the dictionary's items ((key, value) pairs)
`pop(`**`key`**`)`	removes any existing mapping for the given key and returns it (error if key not found)
`popitem()`	removes and returns an arbitrary (key, value) pair (error if empty)
`keys()`	returns the dictionary's keys
`values()`	returns the dictionary's values

You can also use `in`, `len()`, etc.

Dictionaries and tallying

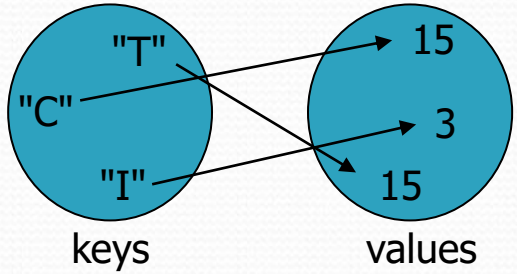
- a `dictionary` can be thought of as generalization of a tallying `list`
  - the "index" (key) doesn't have to be an `int`
  - count digits: `22092310907`




```
# (T)rump, (C)linton, (I)ndependent
```

  - count votes:

```
"TCCCCCCTTTTTCCCCCCTCTTITCTTITCCTIC"
```



items, keys and values

- `items` function returns tuples of each key-value pair
  - can loop over the keys in a for loop

```
ages = {}
ages["Merlin"] = 4
ages("Chester"] = 2
ages["Percival"] = 12
for cat, age in ages.items()):
    print(name + " -> " + str(age))
```

- `values` function returns all values in the dictionary
  - no easy way to get from a value to its associated key(s)
- `keys` function returns all keys in the dictionary

Exercise: Mountain peak

Write a program that reads elevation data from a file, draws it on a `drawing_panel` and finds the path from the highest elevation to the edge of the region and draws that path on the `drawing_panel`.

Data:

```
34 76 87 9 34 8  22 33 33 33 45 65 43 22
5  7  88 0 56 76 76 77 4  45 55 55 4  5
…
```

This data is a different shape. How should we store it?

Lists of lists

- You can put a list in a list

```
list = [[1, 2, 3], [4, 5, 6]]
```

How can you access 2?

```
list[0][1]
```

How can you find the length of the second inner list (`[4, 5, 6]`)?

```
len(list[1])
```

Lists of lists - traversals

- We normally use nested loops to go through a list of lists

```
data = [[1, 2, 3], [4, 5, 6]]
for i in range(len(data)):
    for j in range(len(data[i])):
        # do something with data[i][j]
```

Why does are inner loop go to `len(data[i])`?

List comprehensions

```
[expression for element in list]
```

- A compact syntax that can replace loops that alter lists
  - Applies the expression to each element in the list
  - You can have 0 or more `for` or `if` statements

List comprehensions

```
vec = [2, 4, 6]
result = [3 * x for x in vec]
print(result)                  # [6, 12, 18]

result2 = [3 * x for x in vec if x > 3]
print(result2)                 # [12, 18]

result3 = [3 * x for x in vec if x < 2]
print(result3)                 # []
```

Notice the contents of `vec` do not change

List comprehensions

More than one element can be generated from each element in the original list

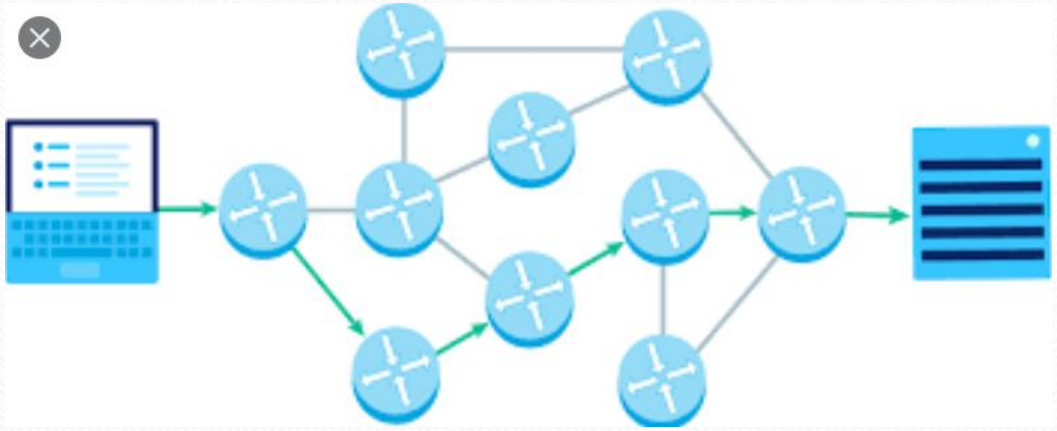
```
vec = [2, 4, 6]
result = [[x, x ** 2] for x in vec]
print(result)                  # [[2, 4], [4, 16], [6,
36]]

result2 = [(x, x ** 2) for x in vec]
print(result2)                 # [(2, 4), (4, 16), (6,
36)]
```

What is Computer Science?

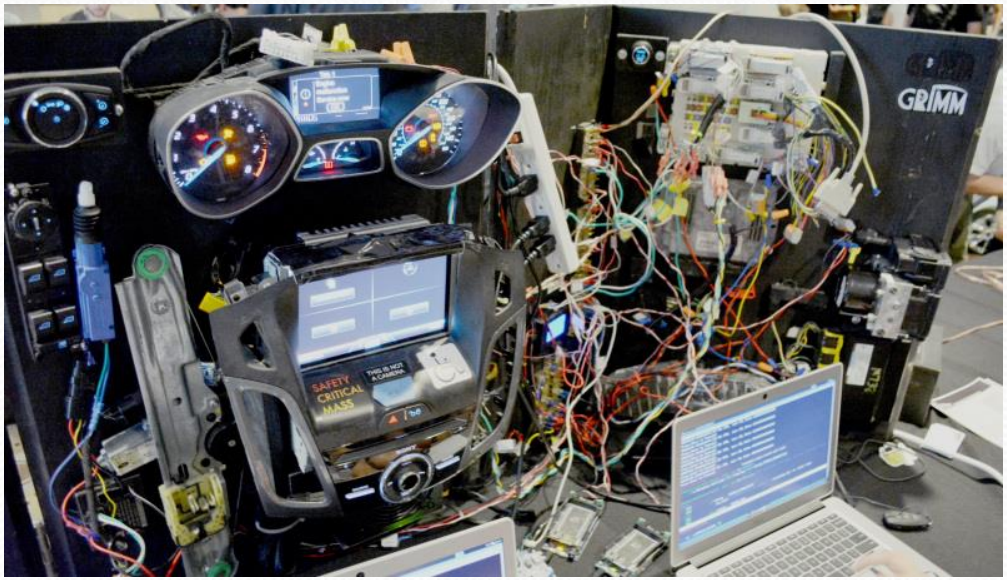
Computer Systems

- Operating systems (Windows, OSX, Linux)
- Networks (the internet, etc)
- Distributed Systems
- Parallel Computing
  - Involves working closer to the hardware



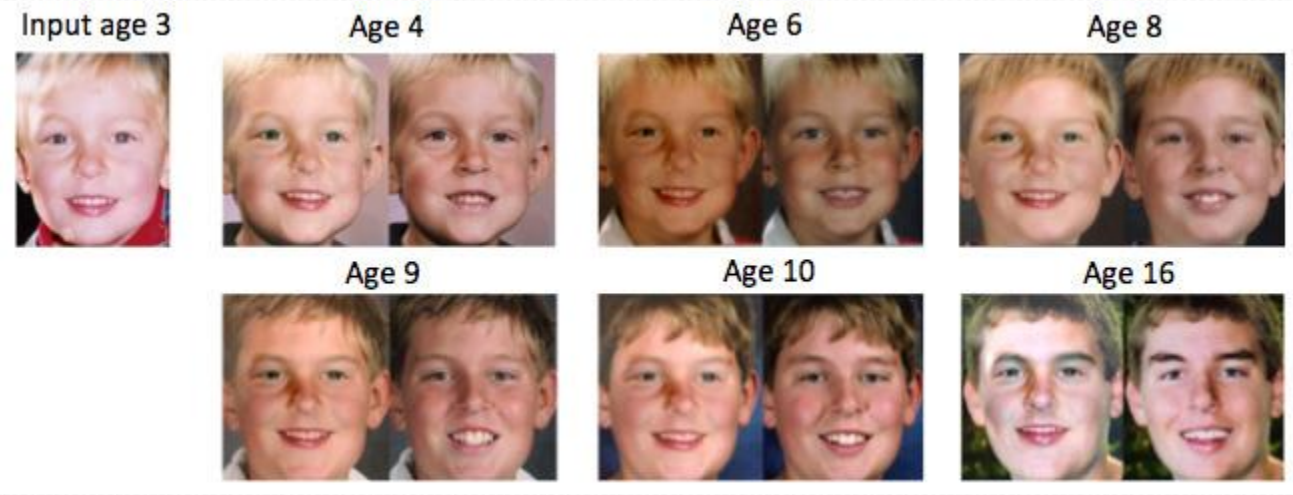
Security

- Almost everything is computerized now
  - Cars
    - Brakes
    - Steering
    - Etc
  - Thermostats
  - …

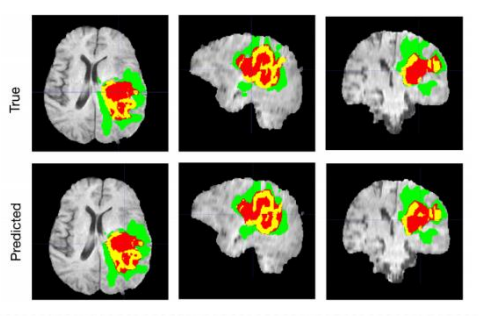


Computer Graphics

- Automatically generating and producing graphics and animations
  - Commonly used in computer games and animated films
  - Also used for simulations like age progression



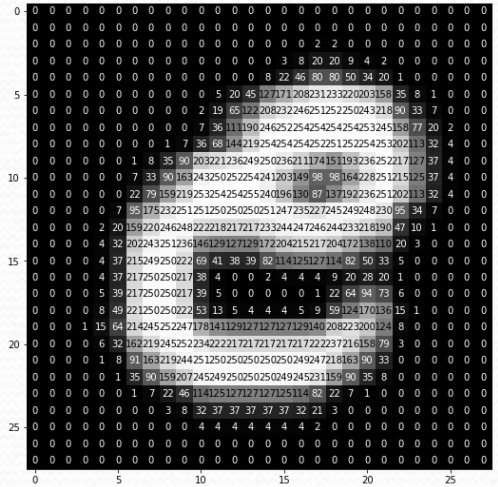
Computer Vision



- Finding items in an image
  - Used by doctors to help read MRIs, Xrays, etc
  - Used by devices like the Microsoft Kinect to sense where you are
  - Used by robots to navigate their surroundings

Artificial Intelligence

- Making a computer appear smart
  - Common applications:
    - chat bots
    - playing a game against the computer
    - cheating detection
    - handwriting detection
    - photo tagging and face recognition
    - spam filtering
    - robotics

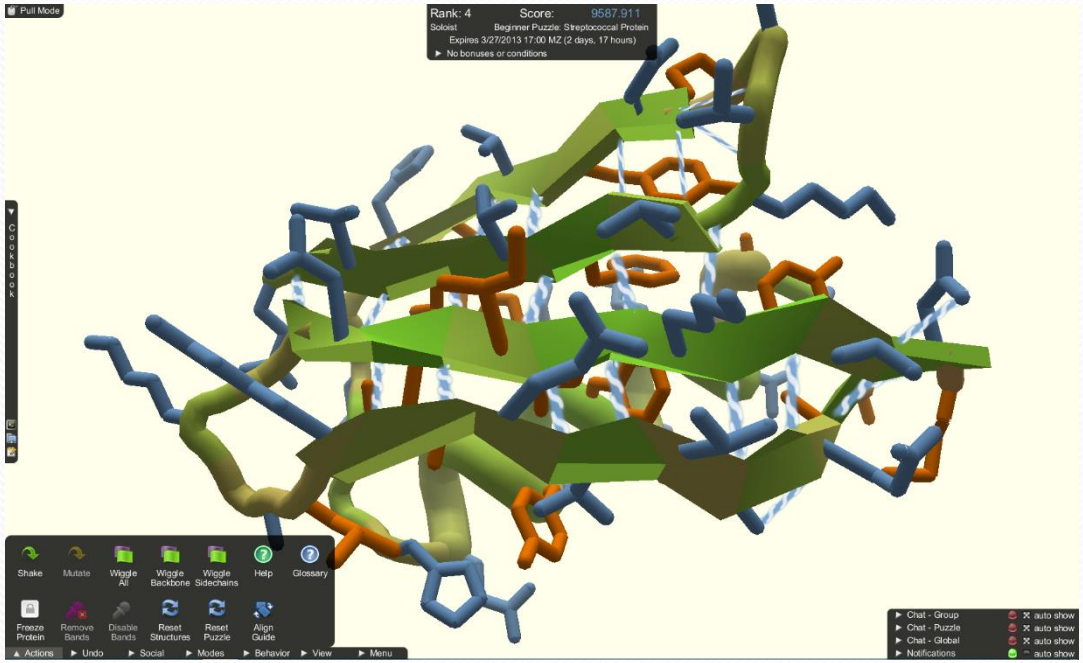


Human Computer Interaction

- How can we design applications that are easy to use and help people?
  - Different interfaces
  - Layouts
  - Designing programs that help people make better choices

Computational Biology

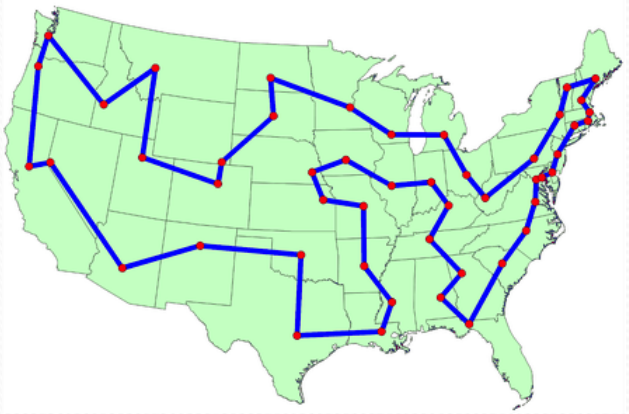
- Using computational modeling and data analysis to solve problems in biology



Algorithms

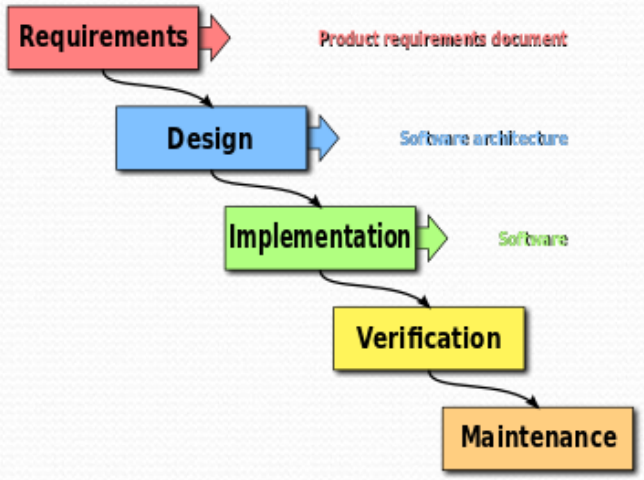
- Coming up with better ways to do things

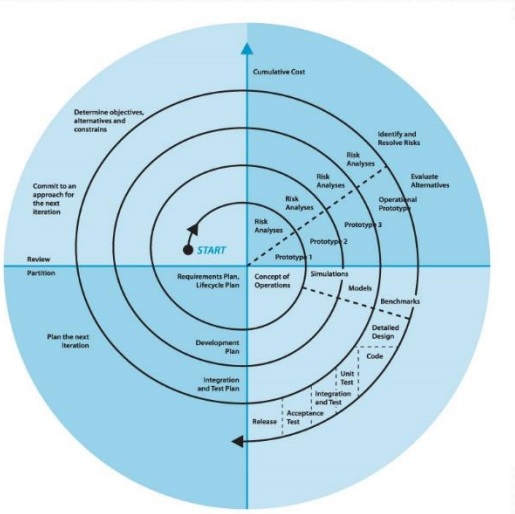
- The traveling salesman problem:
  - A salesperson needs to go around many cities to sell products. What order should they visit the cities in to travel the shortest path?
  - How can we figure this out?



Software Engineering

- Figuring out how to make the process of developing software better
  - How can we avoid bugs?
  - How can we make better designs the from the beginning?





Computing for Good

- There are many applications of computing that help the world
  - Vaccine distribution to resource constrained environments
  - Mobile ultrasound
  - Emergency Alert
  - Access and Accessibility
    - Maps of safe routes for visually impaired
    - Mobile ASL
    - Accessible math

What is hard in computer science?