Design Thinking and String Manipulation

Topics we will cover today:

- String manipulation
- Introduction to Design Thinking and Program Design

Your future in CS

I used to include this on my slides, but since these slides have changed - going to just leave it up here for every notebook. I get a lot of questions about more programming courses, the concentrations, and minors in computer science. Here is a brief reminder.

CS 164 – Next Course In Sequence, also consider CS 220 (math and stats especially)

- CO Jobs Report 2021 – 77% of *all* new jobs in Colorado require programming
- 60% of all STEM jobs requires *advanced* (200-300 level)
- 31% of all Bachelor of Arts degree titled jobs also required coding skills
- 2016 Report found on average jobs that require coding skills paid $22,000 more

- Concentrations in CS:
    - Computer science has a number of concentrations.
        - General concentration is the most flexible, and even allows students to double major or minor pretty easily.
        - Software Engineering
        - Computing Systems
        - Human Centered Computing
        - Networks and Security
        - Artificial Intelligence
        - Computer Science Education.
    - Minors:
        - Minor in Computer Science - choose your own adventure minor
        - Minor in Machine Learning - popular with stats/math, and engineering
        - Minor in Bioinformatics - Biology + Computer Science

Strings

- Are a sequence of characters.
- Like all sequences (lists!) they have indices from 0 until length-1
- Are immutable
    - Fancy word to say, you can't modify a string, but you can get new strings from them!

```
barbarian = "conan the barbarian!"
```

c	o	n	a	n		t	h	e		b	a	r	b	a	r	i	a	n	!
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	16	17	18	19	21

In [ ]:
```
barbarian = "conan the barbarian!"
first = barbarian[0:5]
print(first)
```

```
conan
```

In [ ]:
```
no_last = barbarian[:-1]  ## notice, we are storing the sliced string in a variable
print(no_last)
print(barbarian[:len(barbarian)-1]) #equivalent to above!
```

```
conan the barbarian
conan the barbarian
```

In [ ]:
```
barb = barbarian[-10:]
print(barb)
print(barbarian[len(barbarian)-10:]) #equivalent
```

```
barbarian!
barbarian!
```

A few things to notice:

- variable[start:end] - gives you the 'substring'
  - start is inclusive / includes it
  - end is exclusive / does not include it
- if you omit start, it will assume 0
- if you omit end, it will assume length-1
- len(variable) - gives you the length of any sequence!
  - yes, same command for lists

String Functions and Methods

A method is a function that is 'attached' to a specific String. The format for using them is:

variable.method(parameters)

For example:

k	a	y	a	k
0	1	2	3	4

In [ ]:
```
boat = "kayak"

indexY = boat.find("y")  # .find(str) returns the index location of string within the
print(indexY)

indexA = boat.find("A")
print(indexA) ## woops, case matters! returns -1 if not found
```

```
indexA2 = boat.find("a", indexY) # start looking for a at or after position 2
print(indexA2)

print(boat.rfind("a")) # rfind starts looking from the end, "reverse find"
```

```
2
-1
3
3
```

In [ ]:
```
a_count = boat.count("a")
print(a_count)
```

```
2
```

In [ ]:
```
sound = boat.replace("y", "k")
print(sound) #notice, replace doesn't modify boat, but returns a new string!

boat_large = boat.upper()
print(boat_large)

print(boat_large.lower()) # both .upper() and .lower() are great for setting equivaler
```

```
kakak
KAYAK
kayak
```

In [ ]:
```
dont_talk = "yak" in boat
print(dont_talk)
```

```
True
```

in Operator

Worth talking about. It checks to see if an item is **in** a sequence, and returns true or false

- works on Strings to see if a string contains something (case matters!)
- works in lists to see if a list contains an item!

Lists and Strings

Both sequences, but it is common to want to 'convert' between them.

We do that using `.split` and `.join`

In [ ]:
```
csv = "skeleton,13,12,20".split(",")
print(csv) # it is now a list of strings!

line = "=>".join(csv) # notation is odd, you call join on the string you want to join

print(line)
```

```
['skeleton', '13', '12', '20']
skeleton=>13=>12=>20
```

Exploiting Patterns

- Why does this matter?
- File formats are essentially "patterns" we exploit
  - Why? Because it allows structure to exist
  - We can use that structure
- Being able to find and exploit patterns is extremely helpful
  - Also a good way to look at the world

File Format Example

- HTML files are the basis of webpages
- They have the pattern of

```
<h1>Header</h1>
 <p>paragraph</p>
```

- Notice the "markup" is bracket with the type of info to display.
- A web browser reads the file, and uses that set pattern to help it display the information.

Data Example:

Assume we have a file with the following information:

```
Field One:pH=7.2,P2O5=23,K20=5,Ca=40,S=30,Lat=40.5853N,Lon=105.0844W;Field
Two:K20=6,P2O5=23,pH=7.1,Ca=41,S=30,Lat=40.5852N,Lon=106.0844W
```

There is a pattern to this data/soil test.

```
Field Identifier:stats separated by commas;Field Identifier:stats
```

Knowing this pattern I can use it to write a program that pulls up field information.

In [ ]:
```
sample="Field One:pH=7.2,P2O5=23,K20=5,Ca=40,S=30,Lat=40.5853N,Lon=105.0844W;Field Two

def getSingleSample(data, sampleId):
    sampleStart = sample.find(sampleId)
    sampleEnd = sample.find(";", sampleStart)
    return data[sampleStart : sampleEnd]

print(getSingleSample(sample, "Field One"))
print("--")
## this is also a good case for a split
all_samples = sample.split(";")

for s in all_samples:
    print(s)
```

```
Field One:pH=7.2,P2O5=23,K20=5,Ca=40,S=30,Lat=40.5853N,Lon=105.0844W
--
Field One:pH=7.2,P2O5=23,K20=5,Ca=40,S=30,Lat=40.5853N,Lon=105.0844W
Field Two:K20=6,P2O5=23,pH=7.1,Ca=41,S=30,Lat=40.5852N,Lon=106.0844W
```

Your Turn! (In Class Activity)

Given the string below, complete the function that will return the grade for the course.

For example:

```
find_grade('Amy', 'cs164', data_str)
```

should return **A**

Write the code in steps!

- First see if you can convert the data_str to a lower case string, since case doesn't matter.
  - Same with the other variables passed in
- First see if you can find the String that Amy is the start, or Rory is the start
  - print out the entire string
  - ok, no see if you can find the class
- continue taking it in steps, repeating until you solve it.

As always, in class activities mean one programmer at the table, the rest **guide** the programmer.

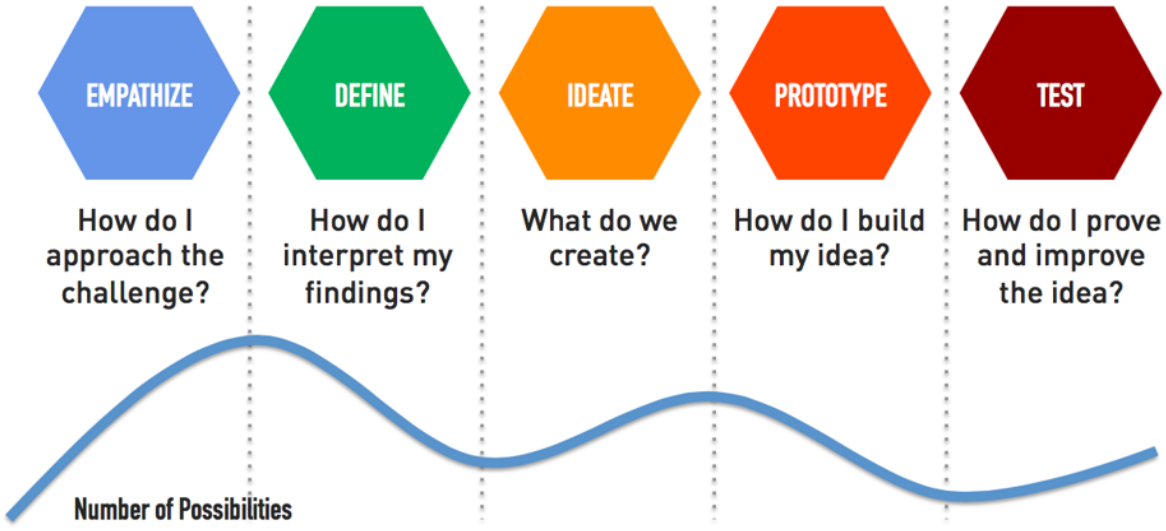
In [ ]:
```
data_str = "Amy:Classes=CHEM107(B),GEO120(C),CS164(A);Rory:classes=CS150B(B),CHEM107(A

def find_grade(student, course, data):
    info = find_student(student, data) ## divide and conquer!
    course_info = find_course(course, info)
    grade_start = course_info.find("(") + 1
    grade_end = course_info.find(")")
    return course_info[grade_start : grade_end]


### these methods are optional, but using them helps you divide the problem up into sm
# maybe try calling these functions in find grade, and print out the results to see wh
def find_course(course, data):
    course_start = data.lower().find(course.lower())
    course_end = data.find(")", course_start) +1
    return data[course_start : course_end]


def find_student(student, data):
    dlower = data.lower()
    info_start = dlower.find(student.lower())
    info_end = dlower.find(";", info_start)
    return data[info_start : info_end]

print("Amy's grade for CS164 is ", find_grade('amy', 'cs164', data_str))
print("Rory's grade for CHEM107 is ", find_grade('Rory', 'CHEM107', data_str))
```

```
Amy's grade for CS164 is  A
Rory's grade for CHEM107 is  A-
```

Design Thinking

- Software Engineering focuses on designing system to solve the problems
  - Like architecture for but for software!
- Design is at the heart of computer science
  - Creative Design
  - Dealing with large systems

    - Problem solving is about designing solutions to those problems
- Design Thinking
    - User Centered Design
    - Human-Centered Design
    - Been around for ~40 years in Computer Science
        - John Arnold 1959
        - IDEO - 1990 coined term



Another way to look at it

- Empathize - Find People
- Define - Look for patterns
- Ideate - Design principles and tasks
- Prototype - Make tangible / code something
- Test - Iterate Relentlessly

This entire process repeats itself not only from the start, but also at each stage.

Practice

- Take 4 minutes to define 6 challenges that are interesting to you.
- 3 Dreams/Things you wish existed and 3 gripes/things that could be better (Challenges!)
- Practice this on a regular basis

> Alice laughed. “There’s no use trying,” she said: “one can’t believe impossible things.”
> I daresay you haven’t had much practice,” said the Queen. “When I was your age, I always did it for half-an-hour a day. Why, sometimes I’ve believed as many as six impossible things before breakfast.”
> - Alice in Wonderland

Next Steps

- Emphasize
  - Talk with others about your ideas
  - Find **diverse** audiences to talk to
    - Talking with your friends and family only introduces unconscious bias
- Define
  - Define your problem before you write code
- Ideate
  - Design your solution before you write code
  - This can be a rough idea
  - Map out your code on paper
- Prototype
  - Start writing!
- Reiterate
  - It is alright to change it / make it incremental!

Defining Problems (and working with labs or other class assignments)

- Defining problems is hard!
  - The more iteration and empathize you do - the better!
  - **Reframe as Questions**
- Every Method, Every program, Every loop, You Write
  - What is your quest?
  - What do you know?
  - What do you need?
  - What can you figure out?