CPS 109 - Lab 4

Agenda

Today we're going to talk even more about functions.

One might say I really love functions.

One might also say I put the fun in functions.

One probably wouldn't say I describe functions with unction (though it could be argued)...

Solution to Last Week's Quiz

Last week the quiz was particularly difficult. However, there were generally 3 key observations to make:

1. When we want to keep track of something it is useful to have a counter variable
2. We usually do not want to modify lists as we iterate over them (consider len())
3. It is useful to start with an empty list and add elements sequentially

Functions

I have already drilled functions into your heads, but let's look at their syntax once more.

```
def backward_string(myString):
```

Here you have "def", the name of your function and whatever arguments you need.

Functions

In Python, functions usually have one or more print or return statements:

```
return new_string
```

```
print(new_string)
```

Functions

Why do we use functions? Well, for a couple of reasons:

1) It saves us from not having to write the same code over and over again.
2) Programs will run very predictably when we call functions.
3) It makes our programs substantially easier to read.

Functions

Functions should do exactly one thing. Examples may include searching a list, or printing some numbers. Functions up until now have had just code plopped in there.

Now, when you make a function, it should only do one thing.

Functions

A simple function (to remind ourselves of function structure):

```
def backward_string(my_string):
    new_string = my_string[::-1]

    return new_string
```

Functions

Now let’s call the function in the main:

```
if __name__ == "__main__":

    inp = "temp"

    while(inp != ""):
        inp = input("Enter a string to reverse:\n")
        print(backward_string(inp))

    print("Good bye~")
```

Functions

Something's weird about how we call print, right?

```
print(backward_string(inp))
```

We're passing a function as an argument for another function! Why are we able to do that?

Functions

Well, remember our function definition on the last slide?

```
def backward_string(my_string):
    new_string = my_string[::-1]

    return new_string
```

Notice how it returns new_string.

Functions

Since the computer knows what instructions to execute and the value of the argument fed into the function, it knows the return value. Meaning that:

```
backward_string(inp) == new_string
```

Functions

What do we do with this information? Well, that means that we can feed functions into other functions as much as we want! This includes functions we've defined.

Nested Functions

```python
# Silly function example

def I_square_things(num):
    return num**2

def I_cube_things(num):
    return num**3

def I_put_things_to_the_exponent_of_6(num):
    return I_cube_things(I_square_things(num))

'''THEY'RE OVERRUNNING THE OUTPOST SAVE US BE-
```

```python
if __name__ == "__main__":
    x = 2
    print(I_square_things(x))
    print(I_cube_things(x))
    print(I_put_things_to_the_exponent_of_6(x))
```

```
4
8
64
[Finished in 51ms]
```

Fun fact: the sixth power was also dubbed the “zenzicube” by the inventor of the = sign

Functions

One more thing about functions before we go through some examples: anything you can write that functions as code (for loops, while loops, if statements, etc.) can (and should!) be put into a function.

Remember: good code is neat code.

“Bin” there done that (A quick review of Binary Search)

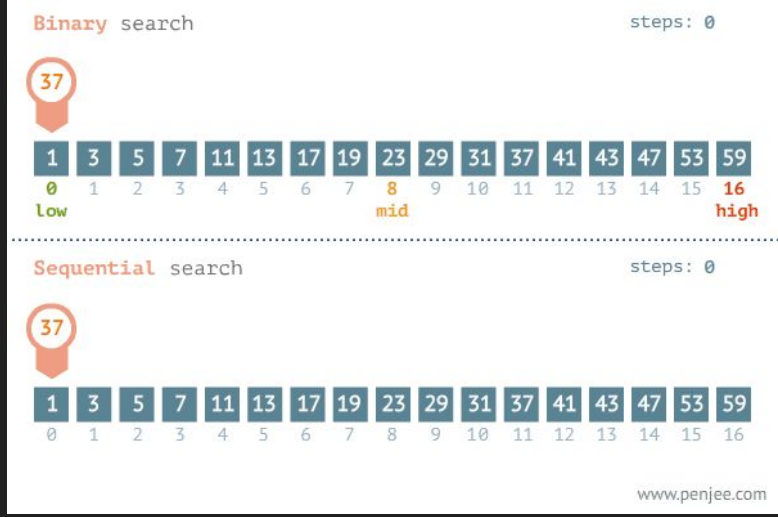
Before we set you off to work on your lab, we’ll give you a quick review of Binary Search.

Different kinds of searches and sorts are very important in computer science, as is the analysis of their runtimes

High Level Steps

1. Set mid to middle of a **sorted** array
   a. If the number at the middle is greater than the number we're looking for then restrict our search to the lower half
   b. If the number at the middle is less than the number we're looking for then restrict our search to the upper half
   c. If the number at the middle is the number we're looking for then return its index (done)
1. Continue 1. until we encounter c. or we have no range left (in which case, the number is not in the list).

To note: By halving our search space at each step, we are much more efficient than in a "naive" sequential search (but this requires a sorted array as input!)



Pseudocode Implementation

```
#Input: The list to search, the element to look for
#Output: The index of the element in the list, OR -1 if it is not in the list
def binary_pseudocode(input_list, element_to_look_for):
    low = 0, mid = 0, high = len(input_list)-1
    while an unsearched range still exists: # What does this mean?
        mid = the floor average of low and high # Recall syntax: //2
        if the item at mid == the element to look for:
            return the index of the item at mid
        if the item at mid > the element to look for:
            update the range to be the lower half of the list (update high)
        if the item at mid < the element to look for:
            update the range to be the upper half of the list (update low)
    return -1 # I.e. the element is not in the list
    # Super cool hint: What are low and high equal to at the end? Are they
    useful?
```