# 15110 Quiz 3

Keerthana & Fiona

# 1D Lists

- A list is a data structure that holds an ordered collection of data values.
- For loops to access elements in a list

Lists share most of their core operations with strings. You can **concatenate** lists together, just like strings.

```
[ 1, 2 ] + [ 3, 4 ] # concatenation – [ 1, 2, 3, 4]
"ABC" + "DEF" # concatenation - "ABCDEF"
```

And you can also **repeat** lists an integer number of times. This works for strings too!

```
[ "a", "b" ] * 2 # repetition – [ "a", "b", "a", "b" ]
"HA" * 3 # repetition - "HAHAHA"
```

We learned about **indexing** and **slicing** last time- those work on lists too.

```
lst = [ "a", "b", "c", "d" ]
lst[1] # indexing – "b"
lst[2:] # slicing – [ "c", "d" ]
```

# findMax()

```
def findMax(nums):
    biggest = nums[0] # why not 0? Negative numbers!
    for i in range(len(nums)):
        if nums[i] > biggest:
            biggest = nums[i]
    return biggest
```

# Strings (and how to use built in functions)

- Strings can be accessed by each character, and within each character you can perform methods.
- Use in to see whether a specific character or part exists in a string (use it as a boolean check)

# Built in functions for strings

```
len(s) # length of a string/list

ord(c) # ASCII number of a character
chr(x) # character associated with the ASCII number

min(lst) # min element of the list
max(lst) # max element of the list
sum(lst) # total sum of elements in the list

random.choice(lst) # picks a random element from the list
```

# 2D Lists

- A 2D list is a list that contains lists within.
- When indexing into 2D lists, you must first index into the original list, and then index further to access specific data values.
- Loop over 2D lists the same way you would over 1D lists (what would be the difference?)

```
cities[2]
cities[2][1]
len(cities)
```

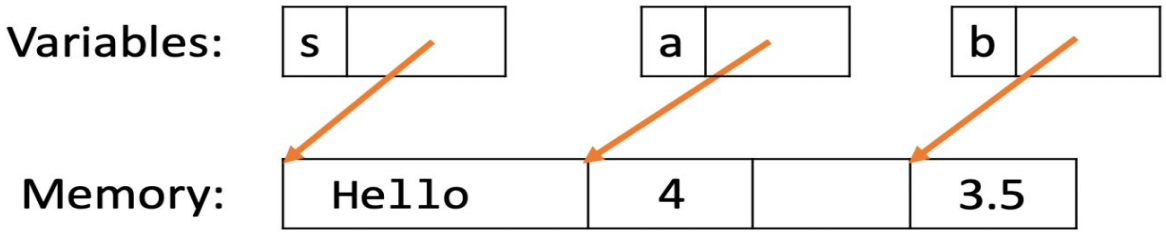
# String Methods

- String methods can be used to either return new values, or to return information.
- Call string methods differently than normal methods
  - s.isdigit()
- In s.isdigit() you would be calling isdigit on s the string
- Use the . to indicate what string you are calling it on
- When using string methods, do not memorize them, there is an API you can use!

# References and Memory

- A reference (often called a pointer) is a specific address in memory. References are used to connect variables to their values.

```
s = "Hello"
a = 4
b = 3.5
```



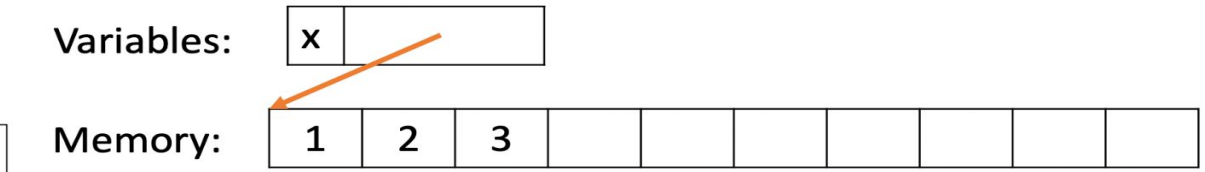
# References and Memory

- Let’s say you have the value s which equals “bye”.
- You make s = s + “sam”
- Now s has a new value which is “bye sam”
- The reference to “bye” no longer exists.
- If you made y = s, now y and s have the same reference to bye sam.

# References and Memory

- When you initialize a list, there is a large chunk of memory allocated to this list, more than what is necessary.

```
x = [1, 2, 3]
```




Technically each index also holds a reference to a new location, but that's out of scope for this course

# Mutable and Immutable Values

- Lists are mutable and strings are immutable. This is important!
- We call data types that can be modified without reassignment this way mutable. Data types that cannot be modified directly are called immutable.
- This is referring to how you do .append to a list, but to change a string, you must create an entirely new string.
- Creating a new list by making it equal to the old one copies the reference as we went over previously.

# Mutable and Immutable Values

- If you share a reference between two lists, and change one of them, both of them reflect the change.
- The reason this happens is because of the reference these two lists share!!

# Destructive vs. Non- destructive

- Destructive approaches change the data values without changing the variable reference. Any aliases of the variable will see the change as well, since they refer to the same list.
- Non-destructive approaches make a new list, giving it a new reference. This 'breaks' the alias and doesn't change the previously-aliased variables.

How do we add a value to a list **destructively**? Use destructive methods - `append`, `insert`, or `extend`.

```
lst = ["A", "B", "C"]
lst.append("E") # add value to the end
lst.insert(0, "foo") # inserts 2nd param into 1st param index
lst.extend(["F", "G"]) # adds multiple elements
```

How do we add a value to a list **non-destructively**? Use variable assignment with list concatenation.

```
lst = ["A", "B", "C"]
lst = lst + ["E"] # note that "E" needs to be in its own list
# warning: 'lst +=' and 'lst = lst +' behave differently!
lst = lst[:len(lst)//2] + ["F"] + lst[len(lst)//2:]
```

# Aliasing

- In Python, aliasing happens **whenever one variable's value is assigned to another variable**, because variables are just names that store references to values.

```
def foo(lst):
    lst[1] = "bar"

x = [1, 2, 3]
print(foo(x)) # when lst is created, it copies x's reference
print(x) # now 2 has been replaced with "bar"
```

# Recursion

To solve a problem recursively:

1. Find a way to make the problem slightly smaller

2. Delegate solving that problem to someone else

3. When you get the smaller-solution, combine it with the solution to the remaining part of the problem to get the answer

# Recursion

- The base case is the one of the most important steps to recursion. It solves the problem without delegating, and it’s the simplest case of the entire recursive process.
- The other case is the recursive case, and this is when you keep going until you reach the smallest problem, at that point which you will reach the base case.

```
def recursiveAddCards(cards):
    if cards == [ ]:
        return 0
    else:
        smallerProblem = cards[1:]
        smallerResult = recursiveAddCards(smallerProblem)
        return cards[0] + smallerResult
```

# factorial(x)

```
def factorial(x):
    if x == 1: # base case
        return 1 # something not recursive
    else:
        smaller = factorial(x - 1) # recursive call
        return x * smaller # combination
```

countVowels()

```
def countVowels(s):
    if _____: # base case
        return _____
    else: # recursive case
        smaller = countVowels(_____)
        return _____
```

# Tower of Hanoi

Recursive solution:

1. Delegate moving all but one of the discs to the temporary platform.
2. Move the remaining disc to the end platform.
3. Delegate moving the all but one pile to the end platform.

# Tower of Hanoi

```
# Prints instructions to solve Towers of Hanoi and
# returns the number of moves needed to do so.
def moveDiscs(start, tmp, end, discs):
    if discs == 1: # 1 disc - move it directly
        print("Move one disc from", start, "to", end)
        return 1
    else: # 2+ discs - move N-1 discs, then 1, then N-1
        moves = 0
        moves = moves + moveDiscs(start, end, tmp, discs - 1)
        moves = moves + moveDiscs(start, tmp, end, 1)
        moves = moves + moveDiscs(tmp, start, end, discs - 1)
        return moves

result = moveDiscs("left", "middle", "right", 3)
print("Number of discs moved:", result)
```

# Searching Algorithms

## Linear Search - Iterative

- in python we can use the **in** operator to check if an item is in a list
  - the **in operator** is implemented with a search algorithm
- analogies for **linear search**
  - uncovering a row of cards one by one to find a specific card
  - checking a stack of books to find a specific book

# Searching Algorithms

## Linear Search - Recursive

- Base Case(s)
  - If the input is an **empty list** we want to return False
  - If the list contains a **single element** that is our target we want to return True
- Recursive Case
  - The **smaller problem** is everything minus the first element in the list

# Searching Algorithms

## Linear Search - Recursive

```
def recursiveLinearSearch(lst, target):
    if lst == [ ]:
        return False
    elif lst[0] == target:
        return True
    else:
        return recursiveLinearSearch(lst[1:], target)
```

# Searching Algorithms

## Binary Search - Recursive

- We can use binary search on a sorted list
  - Start in the middle and eliminate **half of the list** after comparing your target with the middle element
- Analogies for binary search
  - searching for a book in a library - sorted by author's name
  - searching for a house on a street - houses arranged in order

# Searching Algorithms

## PRACTICE - CODE TRACING

- How many times will the function binarySearch() be called after the following code runs?

```
def binarySearch(lst, target):
    print("search called")
    if len(lst) == 0:
        return False
    else:
        midIndex = len(lst) // 2
        if lst[midIndex] == target:
            return True
        elif target < lst[midIndex]:
            return binarySearch(lst[:midIndex], target)
        else: # lst[midIndex] < target
            return binarySearch(lst[midIndex+1:], target)


binarySearch([2,4,5,7,9,14,23], 1)
```

# Dictionaries

- Key-value pairs
  - dictionaries store information in key-value pairs
  - you can access specific values in the dictionary by looking up the key
  - keys **must be immutable!**
- Using loops on dictionaries
  - We can use **FOR EACH** loops when working with dictionaries to loop **directly over the keys** (for key in d)
  - to access the value corresponding to every key we can use d[key] (NOT “key”!)

# Dictionaries

## PRACTICE - CODE WRITING (Part 1)

Write a function meanAge(d) that takes in a dictionary **d** that maps a student's name (a string) to their age (an integer) and calculates the mean age of students in the class.

```
d = {"Bill": 20,
     "Sophie": 35,
     "Millie": 19,
     "Hailey": 18,
     "Freddie": 20,
     "Derek": 21,
     "Peter": 18,
     "Kevin": 22}
```

# Dictionaries

## PRACTICE - CODE WRITING (Part 2)

Write a function newStudents(d, names, ages) that takes in a dictionary **d,** a list of **names, and** a list of corresponding **ages** (guaranteed to be the same length) and adds the name and age to the class dictionary. If the name already exists, do not add the person to the dictionary. Your function should return the updated dictionary.

```
d = {"Bill": 20,
     "Sophie": 35,
     "Millie": 19,
     "Hailey": 18,
     "Freddie": 20,
     "Derek": 21,
     "Peter": 18,
     "Kevin": 22}
```

# Hashing

- Hashtables
  - a list with a fixed number of indexes
  - it stores value(s) in buckets
- Hash functions
  - we assign values to a bucket based on its **hash value** instead of placing it at the end of the list
  - hash functions **map values to integers (this is the hash value)**

# Hashing

- Requirements of a good hash function
  - Given a specific value x, hash(x) must always return the same output i
  - Given two different values x and y, hash(x) and hash(y) should usually return two different outputs, i and j
- Searching a hash table
  - (1) Call the hash function on the value -> this gives you the hash value (an index)
  - (2) ONLY check the bucket with the corresponding index
  - Since there is a “constant” number of elements in each bucket we have an O(1) runtime

# Hashing

- What must be true for us to look up a value in a hashtable in **constant** time?

# Hashing

## PRACTICE - Hash Functions

Put the following words into the hash table below using the hash function **mysteryHash(s)**: “flower”, “christmas”, “winter”, “pumpkin”, “goodbye”

```
def mysteryHash(s):
    if ord(s[0]) % 2 == 0:
        return ord(s[-1]) + len(s) % 10
    else:
        return len(s)//2 + ord(s[len(s)//2])
```

| Hex | Dec | Char | Hex | Dec | Char | Hex | Dec | Char |
|---|---|---|---|---|---|---|---|---|
| 0x20 | 32 | Space | 0x40 | 64 | @ | 0x60 | 96 | ` |
| 0x21 | 33 | ! | 0x41 | 65 | A | 0x61 | 97 | a |
| 0x22 | 34 | " | 0x42 | 66 | B | 0x62 | 98 | b |
| 0x23 | 35 | # | 0x43 | 67 | C | 0x63 | 99 | c |
| 0x24 | 36 | $ | 0x44 | 68 | D | 0x64 | 100 | d |
| 0x25 | 37 | % | 0x45 | 69 | E | 0x65 | 101 | e |
| 0x26 | 38 | & | 0x46 | 70 | F | 0x66 | 102 | f |
| 0x27 | 39 | ' | 0x47 | 71 | G | 0x67 | 103 | g |
| 0x28 | 40 | ( | 0x48 | 72 | H | 0x68 | 104 | h |
| 0x29 | 41 | ) | 0x49 | 73 | I | 0x69 | 105 | i |
| 0x2A | 42 | * | 0x4A | 74 | J | 0x6A | 106 | j |
| 0x2B | 43 | + | 0x4B | 75 | K | 0x6B | 107 | k |
| 0x2C | 44 | , | 0x4C | 76 | L | 0x6C | 108 | l |
| 0x2D | 45 | - | 0x4D | 77 | M | 0x6D | 109 | m |
| 0x2E | 46 | . | 0x4E | 78 | N | 0x6E | 110 | n |
| 0x2F | 47 | / | 0x4F | 79 | O | 0x6F | 111 | o |
| 0x30 | 48 | 0 | 0x50 | 80 | P | 0x70 | 112 | p |
| 0x31 | 49 | 1 | 0x51 | 81 | Q | 0x71 | 113 | q |
| 0x32 | 50 | 2 | 0x52 | 82 | R | 0x72 | 114 | r |
| 0x33 | 51 | 3 | 0x53 | 83 | S | 0x73 | 115 | s |
| 0x34 | 52 | 4 | 0x54 | 84 | T | 0x74 | 116 | t |
| 0x35 | 53 | 5 | 0x55 | 85 | U | 0x75 | 117 | u |
| 0x36 | 54 | 6 | 0x56 | 86 | V | 0x76 | 118 | v |
| 0x37 | 55 | 7 | 0x57 | 87 | W | 0x77 | 119 | w |
| 0x38 | 56 | 8 | 0x58 | 88 | X | 0x78 | 120 | x |
| 0x39 | 57 | 9 | 0x59 | 89 | Y | 0x79 | 121 | y |
| 0x3A | 58 | : | 0x5A | 90 | Z | 0x7A | 122 | z |
| 0x3B | 59 | ; | 0x5B | 91 | [ | 0x7B | 123 | { |
| 0x3C | 60 | < | 0x5C | 92 | \ | 0x7C | 124 | \| |
| 0x3D | 61 | = | 0x5D | 93 | ] | 0x7D | 125 | } |
| 0x3E | 62 | > | 0x5E | 94 | ^ | 0x7E | 126 | ~ |
| 0x3F | 63 | ? | 0x5F | 95 | _ | 0x7F | 127 | DEL |