CPS109 List

- Function calls
- Arithmetic operations like + - // / % * ** ()

Primitive Data Types and Objects - Built In Data Types

```python
str -> String (Primitive)(Immutable)
int -> Integer (Primitive) (Immutable)
list -> List (Object) (Mutable)
tuple -> Tuples (Object) (Immutable)
dict -> Dictionary (Object) (Mutable)
float -> Floats (Primitive) (Immutable)
bool -> Boolean (Primitive) (Immutable)
```

- # Remember that Strings and Tuples are immutable but lists aren't
- Remember these built in functions:

```python
break -> Breaks through a loop
continue -> Skips that specific iteration
round(number, # of decimal places you want)
abs(number)
min(list), min(num1, num2, ...)
max(list), max(num1, num2, ...)
len(list, strings, tuples)
str -> generally used for converting some other data type to a string
int -> generally used for converting some other data type to an int
float -> generally used for converting some other data type to a float
bool -> generally used for converting some other data type to a bool
input(always takes input as a string unless specified to convert to something else)
range(start, stop(excluding), step)
sorted(takes in a list and sorts it)
type(checks the type of some value)
enumerate(list) -> # returns a list of tuples (index, value). Generally for forloops
print()
```

Math methods to remember

```python
math.sqrt() -> Square root of any float or int
math.sin(takes a value in radians)
math.cos(takes a value in radians)
math.tan(takes a value in radians)
math.pow(number, exponent)
math.factorial(value)
math.floor() -> Rounds down a value. 3.75 becomes 3
math.ceil() -> Rounds up a value. 3.1 becomes 4
math.exp(power). # Same as e^x from calculus
math.pi # this is not a method and thats why it does not have the brackets
```

String methods to remember

- string.replace(string to replace, what to replace with, how many times to replace) -> If the number of times is not given, it will replace all of them

```python
string = "Hello, my name is Jude"
# Let's remove all the spaces from this string
string = string.replace(' ', '')
# ' ' means space and '' this means empty quotes so basically nothing
# Final string is
```

```
print(string)
'Hello,mynameisJude'
```

- string.strip() -> Gets rid of all the white spaces on the left and the right side of the string

```python
string = "    Hello   "
string = string.strip()
print(string)
"Hello"
```

- string.split(string to split by) -> whenever it sees that string, it will turn that into a separate element

```python
string = "1 2 3 4 5 6" # Everything is separated by spaces
string = string.split(" ")
print(string)
# ['1', '2', '3', '4', '5', '6']
```

- string.find(strings to find) -> return the index

```python
string = "hello my name is Jude"
print(string.find('my'))
# Result is 6 as the substring 'my' starts from index 6
# If in case the string is not found, then it will return -1
```

- string.capitalize() -> Capitalizes the first letter of the first word

```python
string = 'hello'
print(string.capitalize())
# Prints Hello
```

- string.title() -> Capitalizes the first letter of every word (title case)

```python
string = "hello world"
print(string.title())
# Prints 'Hello World'
```

- These few functions are generally used for each character at a time

```python
string.isalpha() -> checks if the character is an alphabet
string.isdigit() -> checks if the character is a digit
string.islower() -> checks if the character is lowercase
string.isupper() -> checks if the character is uppercase
string.count() -> counts the number of times a substring was repeated
'a' is a lowercase letter
'A' is an uppercase letter
'1' is a digit
'c' is an alphabet
```

- string.endswith(substring goes here) -> checks if the strings ends with that particular substring

```python
string = "hello"
print(string.endswith('lo'))
# Prints True
```

- ".join(list)->Concatenates all elements inside a list. Works only with a list of strings.

```python
["Hello", "my", "name", "is", "Jude"] -> HellomynameisJude
```

- string.upper()->Turns all characters to uppercase
- string.lower()-> Turns all characters to lowercase

Note:

- Remember that while working with strings you can also use fstrings.

```python
name = "Jude"
age = 18
print(f"My name is {name} and my age is {age}.") # Better method than using + to print. Readable
```

- There are also raw strings which basically output exactly what is written inside the string

```python
print(r"Hello \n, my name is Jude\t. Nice to meet you all.")
# This prints
"Hello \n, my name is Jude\t. Nice to meet you all."
instead of
"Hello"
", my name is Jude    . Nice to meet you all."
```

List functions to remember

Let's say the variable name is ls

```python
ls.index(value) -> Get the index of value the first time it appears
ls.count(value) -> Similar to string, returns the count of how many times the value appears
ls.sort() -> Sort the list ascending order
ls.sort(reverse=True) -> Sort the list descending order
```

Note that lists are mutable. So ls.sort() automatically changes the original list. You don't have to set the value to the sorted list unlike strings where would have to set the value to the sorted list. For example:

```python
# Thats how sort would work for list
ls = [1, 3, 2, 10, 9, 4]
ls.sort()
print(ls)

# This is how uppercase would work for string
string = "hello"
string = string.upper()
print(string)

# Notice how for string you have to do string = then whatever u want to change it to.
# But for lists, u just say list.sort() and it will sort the original list
# So strings are immutable and lists are mutable

ls.reverse() -> Returns the same list but reading backwards. Similar to ls[::-1]. It DOES NOT SORT.
ls.append(value) -> Append a value to the end of the list
ls.pop(index) -> Removes a value at a particular index and returns that value as well
# NOTE: BY DEFAULT POP ALWAYS GETS RID OF THE LAST INDEX FROM A LIST
# based on the list from earlier
ls.pop() -> This returns 4
print(ls) -> [1,3,2,10,9]
ls.remove(value) -> Removes the first instance of the value in the list

ls = [1, 3, 2, 10, 9, 4]
```

```
ls.pop(3) -> This would remove the element at index 3. So 10 would be removed
print(ls) -> [1, 3, 2, 9, 4]

ls = [1, 3, 2, 9, 10, 9, 4]
ls.remove(9) -> This would remove first instance of the value from the list
print(ls) -> [1, 3, 2, 10, 9, 4]
```

- ls.insert(index, value) -> Append a value in the list before a particular index

```
ls = [1, 3, 2, 9, 10, 4]
ls.insert(2, 5) -> Inserting the value 5 before index 2
print(ls) -> [1, 3, 5, 2, 9, 10, 4]
```
PYTHON

Tuples

- There are only two functions to remember for tuples
- Let the variable name be tuple

```
tuple.count(value) -> counts how many times a value appeared in the tuple
tuple.index(value) -> Returns the index of the value in the first time it appeared in the tuple
```
PYTHON

Set Datatype

- Denoted with curly braces
- Sets don't have any elements being repeated
- Very similar to sets in math
  - Note that {1,2} == {2,1} would still return True because 1 and 2 and both sets
- Order does not matter when it comes to sets
- When to use sets

Use when:

- You want to eliminate duplicate entries and test for membership a lot.
- Much, much faster than a list!
- **Note:** everything is fast when your data has 10 items in it
- In the wild, you deal in millions and billions of elements.
- Set VS list can make the difference between your code running in a couple ***seconds*** and a couple ***hours***. Not an exaggeration.

```
A = {1,2,2,2,2,1,1,3,4,5}
print(A) -> {1,2,3,4,5} -> Notice how this does not include the duplicates
```
PYTHON

- *Set Functions* - For the following functions we will be using the variable A to describe a set

```
A.add(value) -> This would add an element inside the set
A.clear() -> Clears the entire set which means the final statement would print set()

A.union(B) This would return the union between the set A and the set B
A = A.union({-1, 42}) -> This changes the set to {1,2,3,4,5,-1,42} (order may not be the same)

A.difference(B) -> Does set subtraction by doing A - B
A = A.difference({1,2}) -> This changes the set to {3,4,5}

A.intersection(B) -> Returns a set that contains all elements that are the same between A and B
```
PYTHON

```
A = A.intersection({-1,2}) -> Returns {2} because 2 is the only element inside of A and in {-1, 2}

A.issubset(B) -> Checks if A is a subset of B
A.issuperset(B) -> Checks if B is a subset of A
A.issubset({1,2,3,4,5,6}) -> This returns true
A.issubset({1,2,3,4}) -> This returns False because st is not a subset of {1,2,3,4}

# | this also works as a union symbol
print({1,3,2} | {5,1,2}) -> this prints {1,2,3,5} (order may not be the same)
# & this also works as the intersection symbol
print({1,3,2} & {5,1,2}) -> this prints {1,2} (order may not be the same)
```

A | B **A.union(B)**	Returns a set which is the union of sets `A` and `B`.
A |= B **A.update(B)**	Adds all elements of array `B` to the set `A`.
A & B **A.intersection(B)**	Returns a set which is the intersection of sets `A` and `B`.
A &= B **A.intersection_update(B)**	Leaves in the set `A` only items that belong to the set `B`.
A - B **A.difference(B)**	Returns the set difference of `A` and `B` (the elements included in `A`, but not included in `B`).
A -= B **A.difference_update(B)**	Removes all elements of `B` from the set `A`.
A ^ B **A.symmetric_difference(B)**	Returns the symmetric difference of sets `A` and `B` (the elements belonging to either `A` or `B`, but not to both sets simultaneously).
A ^= B **A.symmetric_difference_update(B)**	Writes in `A` the symmetric difference of sets `A` and `B`.
A <= B **A.issubset(B)**	Returns `true` if `A` is a subset of `B`.
A >= B **A.issuperset(B)**	Returns `true` if `B` is a subset of `A`.
A < B	Equivalent to `A <= B and A != B`
A > B	Equivalent to `A >= B and A != B`

Dictionaries

- Dictionaries are data types that have a key and a value
  - Key works as the *index* - Can be any data type
  - Value works as the *value* at that index - Can be any value as well
- Uses these brackets to initialize {}
- Generally used when you want to access a value using another keyword
  - Count how many times each element was repeated inside a list
- Key and value are written in this format

```
dct = {'name':'Sara', 'age':19, 'bf':'Jude', 'school':'TMU'}
# Here the name, age, bf and school work like the index
```

PYTHON

```
# So if we want to access your name we would type
print(dct['name']) -> This outputs 'Sara'
print(dct['age']) -> This outputs 19
print(dct['bf']) -> This outputs 'Jude'
print(dct['school']) -> This outputs 'TMU'
keys = list(dct.keys()) -> This way you would get all the keys
values = list(dct.values()) -> This way you would get all the values
print(keys) -> This prints ['name', 'age', 'bf', 'school']
print(values) -> This prints ['Sara', '19', 'Jude', 'TMU']

dct.items()-> Returns a list of tuples with key and value inside the tuple
print(dct.items()) -> Prints [('name', 'Sara'), ('age', 19), ('bf', 'Jude'), ('school', 'TMU')]

dct.pop(key) -> Gets rid of a key and returns its value
print(dct.pop('school')) -> Prints 'TMU'
```

Tips - Based on Questions

- Always double check what data type you are working with
- When using for loops, make sure you know whether you are working with indexes or values
- For example: It says add every element in the odd index then you know you have to work with indexes. So you would have to use ```

```
variable_name[index]
```
PYTHON

- If you are working with just the values then

```
for value in list_name/string_name/tuple_name:
    do something with value
```
PYTHON

- If you have a list of lists or a list of strings and you need to go through each character at a times. Generally you would do this
- List of strings, list of lists, list of tuples (That is if you do not know the size)

```
list_of_strings = ["Hello", "my", "name"]
for word in list_of_strings:
    for character in word:
        do something
```
PYTHON

- List comprehension Syntax - This is the general syntax
- Value_to_append can be anything you want

```
[value_to_append for i in range()/list/string/tuples]
[value_to_append for i in range()/list/string/tuples if condition]
[value_to_append if condition else value_to_append for i in range()/list/string/tuples]

# See below to see how things are formatted. Round brackets rememble the divide
[(value_to_append if condition) (else value_to_append) for i in range()/list/string/tuples]

# Chained List Comprehension
[value_to_append for i in range(any number) for j in range(any number)...]
# This would run (number * number) times.

# Creating a list of list using list comprehension -> Called Nested List Comprehension
[[value_to_append for j in range(num)] for i in range(num)]
# Lets do this one for example
[[j for j in range(5)] for i in range(3)]
# This would create this list
[[0,1,2,3,4], [0,1,2,3,4], [0,1,2,3,4], [0,1,2,3,4], [0,1,2,3,4]]
```
PYTHON

- Whenever you are creating a variable, be aware of the scope

```python
if x > 5:
    w = 20
    z = 50
print(w, z)
# Here the print function will have an error as the values dont exist outside of the if condition that is why they are inde
# Instead do this
if x > 5:
    w = 20
    z = 50
    print(w, z)
# This way you wont get an error
# You can also swap the print with a return and it will work just fine
```

- If a question says use some type of default values inside a function. Then do this

```python
def function(a=20, b=30)
    return a+b
# function() would return 50
# function(10, 20) would return 30
# function(a=10) would return 40

Lets say we needed a function that needs to have only one default value
Then this is how it would work

def function(a, b=30)
    return a + b

# function() this would give an error as 'a' does not have a value
# function(10) would return 40
# function(10, 20) would return 30
```

- if you ever need to use **global variables** inside a function this is how you would do it.
- Global variables are variables that are not written inside a function

```python
nums = 10
def function():
    global nums
    nums = 20

print(nums) -> This would print 10
function()
print(nums) -> This would print 20 as we changed the value of the nums variable using the function
```

- Remember that **lists** can be sliced the sam way **strings** can be sliced

```python
string[start:stop:step] # This is the general format
string = "Hello"
print(string[2:]) -> Prints 'llo'
print(string[1:4]) -> Prints 'ell'

# List slicing has the same format
list[start:stop:step]
list = [1,2,3,4,5,6,7,8]
print(list[2:6]) -> Prints [3, 4, 5, 6]
```

- Always check if the order of operations make sense
- Remember that any value in python other than an **empty string, empty list or 0** in python is **True**
    - Not the most important point but just a thing to remember incase it pops up

```python
if 1 # is the same as writing
if True
```

```
# Same way
if 0/""/[] # is the same as writing
if False
```

- Some things with tuples

PYTHON

```
# When given a tuple you can use something called **De-structuring** of a tuple
# When we create a variable to represent a whole tuple like this
tup = (1, 2, 3, 4)
# We have to print each of them separately by doing this
tup[0], tup[1], tup[2], tup[3]
# Instead we can destructure the tuple by doing this instead if we know the size of the tuple
a,b,c,d = (1,2,3,4)
# Then we can just put
a, b, c, d # and it will print the values normally

# There is another way to do this as well lets say you want only the values for the first two elements and want to leave th
a, b, *c = (1, 2, 3, 4)
# Here
a = 1
b = 2
c = (3, 4)

# You can also do this
a, *b ,c = (1, 2, 3, 4)
a = 1
b = (2, 3)
c = 4

# When creating a tuple with one element remember to put the comma at the end
# Otherwise it will just be a regular integer
tuple = (2,)
```

- This is how you would generally use enumerate

PYTHON

```
list = ["Jude", "Loves", "Sara", "A", "Lot"]
for index,value in enumerate(list):
    print(f"{index}:{value}")
# This would print
0:"Jude"
1:"Loves"
2:"Sara"
3:"A"
4:"Lot"
```

While loops

PYTHON

```
# This is the general syntax for while loop
while condition:
    do something
# Remember that the condition needs to be true for the while loop to continue looping

# Example
list = [1, 3, 4, 5]
i = 0
while i < len(list):
    print(list[i])
    i+=1
# This would print
1
3
4
5
```

This is when to use for loop or while loop

`for` loop:

- Driven by an iterable.
- Sequence, range, etc.
- Continues iterating until its iterable runs out of values.
- Good when the number of iterations is known (or can be known) a priori

`while` loop:

- Driven by a condition
- “Are we there yet?”
- Iterates until condition is false
- Randomness, input validation
- Good when the number of iterations is not (or cannot) be known a priori

Always remember: “can be known” does not preclude infinite loops, and “cannot be known” in no way *implies* infinite loops.

- We may iterate zero times. Condition is initially false? Iterable is empty?
- “*When there is nothing to do, the best thing to do is nothing*”

- Writing else after a loop

```python
while condition:
    do something
else:
    do something # NOTE: THIS WILL ONLY EXECUTE WHEN THE LOOP DID NOT BREAK OR RETURN
```

Random library

```python
import random
# To generate a random number within a certain range, do this
random.randint(1, 5) -> This will generate a random integer between 1 to 5 both inclusive
random.random() -> This will generate a random value between 0 to 1 without including one and will include decimals
```

- A test question would be to check if your string or list given is a palindrome

```python
# Simplest way to solve that is checking if the forward is the same as the backward
# We will do that using string slicing
string = "racecar"
if string == string[::-1] string[::-1] reverses the string
    return True
else:
    return False
```

Hints

- Integers or floats do not have any index
- **Remember these**
- *n%10 gives you the last digit*
- *n//10 gets rid of the last digit*

```python
# This is the only odds example from the pdf
if n == 0:
    return False
while n != 0
```

```
    if (n%10)%2 == 1: # Gets the last digit and checks if it is an odd number
        n = n//10 # Gets rid of the last digit
        continue
    else:
        return False
return True
```

".join(list) or string.join(list)

PYTHON

```
lst = ['two', 'three', 'one']
tpl = ('two', 'three', 'one')
string = "twothreeone"
spacer = ', '
print(''.join(lst)) ->  Prints twothreeone
print(spacer.join(lst)) -> Prints two, three, one

print(''.join(tpl)) -> Prints twothreeone
print(spacer.join(tpl)) -> Prints two, three, one

print(''.join(string)) -> Prints twothreeone
print(spacer.join(string)) -> Prints t, w, o, t, h, r, e, e, o, n, e
```

File I/O

- For reading from a file or writing or appending to a file
  - *Reading* means read every line from a file
  - *Writing* means rewrite the whole file
  - *Appending* means writing to the bottom of the file without changing anything above
  - **(RAW)**

PYTHON

```
# Best practice for opening a file is this
with open("filename", "choose one of raw") as file ('''file here is a variable'''):
    do something here
# That method closes the file automatically

# Another method of opening a file is
file = open("filename", "r/a/w")
# But while doing this, also make sure that you are closing the file
file.close() -> This method will close the file

# Assuming your file is in the same file is as this python project
with open('filename','r/a/w') as f
# If the file is inside another folder thats still within the same python project
# Then do this
with open('foldername/filename', 'r/a/w') as f
# For the following functions, assume that we are using f as the variable for our file
f.readline() -> Reads a line from a file
f.read(number of characters) -> Reads either whole file(default) or the number of characters mentioned
f.read(number) -> Reads the number of characters specified
f.seek(number of characters, mode)-> If you want to move back or forward a few characters
f.readlines() -> Returns all the lines in the file as a list of strings
f.write(string to write) -> Write to a file (make sure to include the "\n" if you want a new line)
f.close() -> Close a file
f.name -> Gives the name of the file
f.closed -> Tells if the file is closed or not
f.mode -> Tells if the file is in Read Write or Append mode (RAW)
# Example File: File name = 'text.txt'
'''My name is Jude
   Her name is Sara
   And I really really really love Sara
   I love her a lot
   She means the world to me'''
# Lets say we will run this function on this particular file
def readFile():
    with open('text.txt', 'r') as f:
        print(f.readline()) -> Prints 'My name is Jude'
        print(f.readline()) -> Prints 'Her name is Sara'
```

```
        print(f.readline()) -> Prints 'And I really really really love Sara'
        f.seek(1) -> Sets the start position of the file to line 1 again
        print(f.readline()) -> Prints 'Her name is Sara'
        print(f.readline()) -> Prints 'And I really really really love Sara'
        print(f.readlines()) -> Prints ['I love her a lot', 'She means the world to me']
        # Notice how this only has all the files that were not read already
```

Recursion Tips to Remember

- Always contains a base case
- Contains a recursive case which calls itself
- The last part of recursion is always the first thing that executes
  - For example: Think of a stack of plates
  - If you keep on stacking plates one on top of another
  - The last plate is the also going to be the first plate that you take out
  - Last In is always First Out
- Your code must make progress towards the base case
- When your code is supposed to return something make sure you return something in each if condition that you have

PYTHON

```python
# For Example
# This is a recursive function that counts how many elements are there in a list/string/tuple
def count(items):
    if not items:
        return 0
    return 1 + count(items[1:]) -> This line adds one then gets rid of the first element
# Notice how the return statement is there because you want your code to return that 1+ that you are constantly doing
# Otherwise even though your code would work it will return None as the return statement is not there on the recursive call
```

How to reverse a list using recursion

PYTHON

```python
def reverse_list(items):
    if items == []:
        return items
    return [items[-1]] + reverse_list(items[:-1])
print(reverse_list([1,3,2,4]))

# This is the walkthrough for how the recursion is being called
[4] + reverse_list([1,3,2])
      [2] + reverse_list([1,3])
            [3] + reverse_list([1])
                  [1] + reverse_list([])
                        + []
[4]+[2]+[3]+[1]+[] # This is how the recursive steps add
[4,2,3,1] -> # This is the result

# Another way of solving it is this (writing the add part after the recursive call)
def reverse_list(items):
    if items == []:
        return items
    return reverse_items(items[1:]) + [items[0]]

# This is the walkthrough for the recursion is happening here
reverse_items([1,3,2,4])
    reverse_items([3,2,4])
        reverse_items([2,4])
            reverse_items([4])
                reverse_items([])
                reverse_items([])->[]
            reverse_items([4])->[4] +
        reverse_items([2,4])->[2] +
    reverse_items([3,2,4])->[3] +
reverse_items([1,3,2,4])->[1]
[]+[4]+[2]+[3]+[1] = [4,2,3,1]
```

Exception Handling and Assertion

- These are some of the exceptions we have seen so far
  - **StopIteration** -> *When we are out of things to iterate over*
  - **NameError** -> *If a variable is not defined*
  - **TypeError** -> *Wrong type given*
  - **IndexError** -> *Index out of range or not existing*
  - **ValueError** -> *Couldn't Produce a proper value*
  - **ZeroDivisionError** -> *When you divide by zero*
  - **Exception as e** -> *Takes care of all other cases*
- An exception is NOT a return value.
  - We can either raise or return or just do a print statement
  - Raise is like return, you can only do it once in a function
- Try except makes sure the program does not crash
- You can handle multiple exceptions
- When to raise?
  - You raise when your function does not meet the post-conditions(function promises to achieve and guaranteed to have happened once the function is executed)

```python
# Printing the strings are optional
try:
    do something
except ExceptionName:
    return ExceptionName("Message to print")
except Exception as e:
    return e("Something went wrong")
raise ExceptionName("Print Something") -> Raise is like returns but it returns an error instead
```

- **Assertion**
  - Do nothing if a condition is true
  - Throw assertion error if the condition is false
  - Auto grader uses assertion

```python
x = 42
assert(x==42)-> Does nothing
assert(x==12)-> Throws assertion error
```

Memory

- Within the **sys** module

```python
import sys
sys.getrefcount(variable) -> The number of names referring to that object
sys.getsizeof(variable) -> How much space its taking
```

- Slack Space
  - Dictionaries, lists, sets and more maintain slack space
  - It means they are stored using more memory than needed
  - This anticipates the growth of the structure making it much faster

Search and Sort (Memorise These)

- **Bisection Search - Big-O(log n)**

```python
# This is a demonstration of the bisection search as a function
# Finding the square of the number and use either the epsilon given by the user or default
def bisersearch(number, eps=0.001):
```

```
    high = number
    low = 0
    guess = (low + high)/2
    while abs(guess**2 - number) > eps:
        if guess**2 > number:
            high = guess
        else:
            low = guess
        guess = (low + high)/2

# This is a basic square root using a while loop in a regular function
x = number
low = 0
high = x
eps = 0.001 # Delta Value
guess = (low + high)/2
while abs(guess ** 2 - x) > eps:
    if guess**2 > x:
        high = guess
    else:
        low = guess
    guess = (low + high)/2
```

- Selection Sort - Big-O(n^2)
  - Demonstration of Selection Sort Video, click here
  - Demonstration of python code

PYTHON

```
def selection_sort(items):
    for i in range(len(items)): # Go through the list
        best_index = i # First store the current index
        for j in range(i+1, len(items)): # This loop finds the best after the current index
            if items[j] < items[best_index]: # < if ascending order and > if descending order
                best_index = j
        temp = items[best_index]           # { These lines can also be written as
        items[best_index] = items[i]       #  items[i], items[j] = items[j], items[i]
        items[i] = temp                    # }
    return items
```

- Insertion Sort
  - Demonstration of Insertion Sort Video
  - Demonstration of python code

PYTHON

```
def insertion_sort(items):
    for i in range(1, len(items)): # Start from index 1
        current = items[i] # Save the current value
        current_in = i # Save the current index
    # Keep swapping back until we reach the beginning of the list or encounter a value smaller
    # than the current value
    # Use < for ascending and use > for descending order between current '<>' items[current_in - 1]
        while current_in > 0 and current < items[current_in-1]:
            items[current_in] = items[current_in - 1]
            current_in -= 1
        items[current_in] = current
    return items
```

- Interpolation Search or Binary Search - Big-O(log n)
  - Similar to bisection search and is used for binary search
    - Divide and Conquer algorithm (cuts the problem in halves every time)
    - Demonstration Video
    - Code Video

PYTHON

```
def binary_search(items, key):
    low = 0
    high = len(items)-1
    while lo <= high: # This is the condition for binary search to continue happening
```

```
        middle = (high + low)//2 # Since you are working with indexes, you have to do floor division
        if key < items[middle]: # If key is smaller look towards the left side
            high = middle-1
        elif key > items[middle] # If key is larger look towards the right side
            low = middle+1
        else: # If not those 2 cases then that means its equal so return the value
            return middle
    return None # If item not in list then return None
```

- Sequential search Big-O(n)
  - Just going through one element at a time and check if we have found the desired value
  - Regular for loop on a list

Objects and Classes

- Classes are seen as a blueprint and Objects are seen as the result of that blueprint
  - For example if the car blueprint is a class
  - And the car itself is the object
- Almost everything in python is an object
- Classes allow us to encapsulate state (value) and behaviour (methods)
- Objects can have values and methods
- Why Classes?
  - Helps you write cleaner code where properties and behaviour of a certain thing are kept inside a class
    - For example: A class of cards
- Defining a class

PYTHON
```
class Card:
    __vals = {'Ace': 1, 'Two': 2, 'Three': 3, ...}
    def __init__ (self, rank, suit): # This method is called automatically when a class is created
        self.suit = suit
        self.rank = rank
    # This method is called automaticall when the print statement is called on the class
    def __str__ (self):
        return f"{self.rank} of {self.suit}"
    # Syntax for calling the variables inside if the variable start with the __ otherwise you have
    # to call it like self.variableName
    def get_value (self): # This function returns the value of a card
        return Card.__vals[self.rank]
    def outranks (self, other): # The other keyword here also represents and card class
        if other == None:
            return True
        return self.get_value() > other.get_value()
```

- Access Control

PYTHON
```
# __ before a variable name means its hidden from the outside world (mangling)
# It is not private but just harder to get to
```

Concept Questions

- General info about iterators and iterables
  - range() is an iterable (also called a lazy list or lazy sequence)
  - lazy is a term that refers to not doing something until explicitly needed/requested.
  - list is an iterable
  - files are an iterable
  - tuples are an iterable
  - strings are an iterable
  - Dictionaries are an iterable
  - iterators traverse over iterables
  - get an iterator from any iterable object using the iter() function

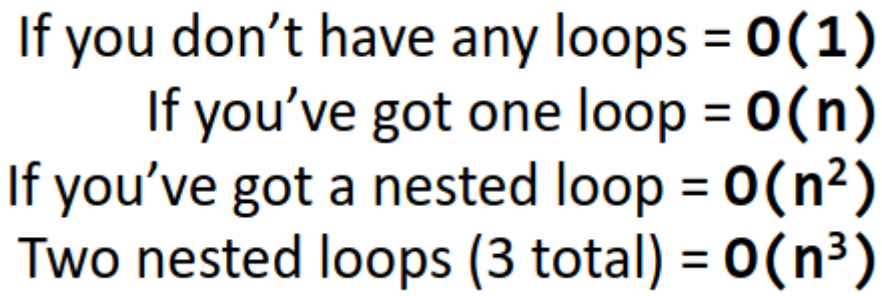
- **use the next() function to go through the elements one at a time**
- **calling next() and at the end of an iterable will throw the StopIteration Error** (functions like file.readline() dont have such issues)
- **iterators added with range shows their true power (because the values does not have to stay loaded into memory)**
- *DOWNSIDES of ITERATORS*
  - *no indexing or slicing*
  - *no rewinding or going back*
  - *strictly goes in sequential order*
  - *you need to produce the 9th element before seeing the 10th*
  - *len() function cannot be used*

PYTHON
```
# A simlpe way to print everything from a file is
for line in file:
    print(line)
```

- General info about iterators and generators
  - **Generators are used to produce custom value in a lazy fashion (for example: Fibonacci, squares or primes)**
  - **Generators are just functions that act like a lazy sequence**
  - **A generator is a function that uses the keyword yield instead of return to produce a result.**
    - each yielded value is the *next* value produced by the iterator
  - **A generator is an iterator**
  - **A generator's namespace and execution state are saved between each yield call**
  - **Returning within a generator function is also legal but it is based on if there are certain conditions that a program is telling you to implement**
  - **A stream of objects can be arbitrarily long, or even infinite and because of iterators and generators, we are still able to read through them in the computer without loading all of it into memory**
- *Simple generator*

PYTHON
```
# This program generates all odd numbers upto 10
def odd_nums(limit):
    num = 1
    while num <= limit:
        yield num
        num += 2
for n in odd_nums(10):
    print(n)
```

- **What is slack space**
  - *Almost all data types in python use slack space*
  - *Strings are one of the only one that increases every time a new character is added so its not a part of slack space*
  - *It means they are stored using more memory than needed*
  - *This anticipates the growth of the structure making it much faster*
- **Big-O notation**



Common Big O Growth Rates	
Big O Expression	**Name**
$O(1)$	constant
$O(\log(n))$	logarithmic
$O(n)$	linear
$O(n^2)$	quadratic
$O(n^3)$	cubic
$O(2^n)$	exponential
$O(n!)$	factorial

- Also make sure to check out the objects and classes and files section to check for some more conceptual questions [sorry I could not finish writing it all here :( ]

General Practice Questions

- Almost palindrome

PYTHON
```
def is_almost_palindrome(word):
    for i in range(len(word)):
        w2 = word[:i] + word[i+1:]
        if w2 == w2[::-1]:
            return True
    return False

# Lets walk through an example here
# Lets say the example word is sirs -> Should be true as a palindrom would be sis and srs if we remove the r
# The idea is to remove a character and see if the word is a palindrome
word = 'sirs'
# In the for loop we create a new temporary word called w2 to create a string with a missing character
When i = 0,
w2 = word[:i]->'' + word[i+1:]->'irs' -> This gives 'irs'
# That would be a no so it does not pass the if check
# Lets check the next case
When i = 1,
w2 = word[:i]->'s' + word[i+1:]->'rs' -> This gives 'srs'
# This would pass the if check so the code would return True
# If the entire for loop is done and we still cannot find a palindrome
# Then the code should return False hence the last line of the code
```

- Transposing a matrix

PYTHON
```
# [
#. [1, 0, 0, 0, 0],
#  [0, 1, 0, 0, 0],
#  [0, 0, 1, 0, 0],
#  [0, 0, 0, 1, 1]
```

```python
# ]
# Based on this diagram, we can see that row is each element in the original list
# And the columns are a length of the element inside a list
# For example a row would be considered [1, 0, 0, 0, 0]
# And the number of columns would be considered the length of this particular row
# For example len(matrix[0]) would print 5 which is the number of columns so the n value
# len(matrix) would print the number of rows so the m value

def transpose(matrix): -> # Recall that a matrix is a 2D list
    tp = [] # Holds the transposed matrix
    n = len(matrix[0])
    m = len(matrix)
    for column in range(n):
        column_to_row_matrix = []
        for row in range(m):
            column_to_row_matrix.append(matrix[row][column])
        tp.append(column_to_row_matrix)
    return tp
```