## DECISION MAKING- IF/ELSE statement: Date/Time Types of Data: #Create a datetime object (from datetime import datetime, time, date ) Integer (23), Float (23.4), String ("hello"), Boolean (TRUE, FALSE) Print to Screen: mydt = datetime(int(year),int(month),int(day),int(hour),int(min),int(sec)) If < condition>: Finding out what data type my variable "x" is and the length: <do something> #This is how to comment #Isolate time and date components Converting to different data types: mydate = mydt.date() mytime = mydt.time() Elif < condition > : Store user input: str(x) #convert variable x into string int(y) #convert variable y into integer < do something> var = input("please enter: ") #datetime arithmetic float(z) #convert variable z into float add a day to your mydt:(use timedelta, remember to import it) mvdt + timedelta(davs=1) <do something> String Concatenation: Essentially: +/- timedelta(days=x, hours=x, minutes=x, seconds=x) combine strings and variables together: Define a variable: x ="string1+ str(age)" + "words" DECISION MAKING- While statement: #Replace a date or a time: mydt = mydt.replace(hour=12) b = 4String Formatting While < condition > : t = a + b< Do something> Use place holders { } in our string - to mark where Try/Except variables should go to make it easier visualize output. Below are two ways to format: DECISION MAKING- For statement: a,b = "Bob","Ali" #One liner assignment <try this which could give an error> $msg = f'We combo \{a\} and \{b\}'$ except <the expected error>: Handling For i in mv list: msg2 ="We combo {} and {}".format(a,b) <do this to show error was caught> < Do something with variables #you can use sys.exit() to stop the script Functions: Class: Define a function: Note: "i" will store the element DECISION MAKING-Def myfunction(): A class can have a set of variables and methods. Boolean IF: <Do something> #e.g #is hot --> boolean $mv \ list = [1, 2, 3]$ myfunction() # call function Inside a class, we call variables: attributes and functions: methods. For i in my list: If is hot: print(i) #do this if true This will print: 1 2 3 The class is a blueprint we use. In our program we create a copy, an Example: "object" of that class. that object has access to the attributes and def myfun(a,b): **Python Conditions** dump = a + bmethods. Here is an example of a class with attributes and methods. Loop through a range of values List comprehension ans = myfun(1,2)for i in range(3): #Define some attributes: print("answer: {}".format(ans) print(i) def init (self, name, weight): We can construct a new list from an old one This will print: 0 1 2 prints --> answer: 3 self.name = name New List = [ < do this > for i in my list]self.weight = weight Do some calculation with respect to "i", for every "i", for i in range(1,4): in this predefined list. #Define some methods: print(i) **Data Structures** def myname(self): This will print: 1 2 3 Eg: Classes/functions print("My name is: " + self.name) $my \ list = [1,2,3]$ def myweight(self): New List = [i\*2 for i in my list]List unpacking: print("My weight is: " + self.weight) # New List becomes [2,4,6] Dictionary: Assign variables to elements in a list/tuple: We have a key-value pair in a dictionary, like so: Outside the class, in the main program, create a cat object: Coordinates = (1,2,3)Kev Value x.v.z = coordinates 500 "John doe" List: cat1 = cat("Andy","10")"Alice walt" store a collection of data (of any datatype) #Call a method defined in the class: Access elements in list: cat1.myname() mylist = [1,2,"hey"] # access an element: mylist[i] for i in list: Use a dictionary to store the above information. # "i" contains element Add item to end: Define an empty dictionary: Notes: mylist.append(120) $dict = \{\}$ 2D lists To create the class, we need a constructor containing the attributes Add value at a certain index: mymatrix = Create a dictionary: followed by any functions. mylist.insert(<index>, <value>) note left side are the keys which must be string or int [4,5,6]. dict = {"500": "John doe", "501": "Alice walt"} Remove or "pop" an item at a given index To define attributes, you need this \_\_init\_\_(). Inside the brackets we [7,8,9] mylist.pop(i) need "self" followed by a list of all the attributes. The class will accept Look up something in the dictionary: Access elements in 2D list: parameters to be used within its collection of methods **Tuples:** print (dict ["500"]) # this prints:"john doe" for row in mymatrix: Same as lists except they are immutable- cannot for element in row: "self." Is attached to an attribute that you are referencing from the class change print element Update an entry in dictionary: Mytuple = (1,2,3) variables, so you need to always write self.<name\_of\_variable>

dict["500"] = "Harry Holland"