**Python Journal Template**

**Directions:** Follow the directions for each part of the journal template. Include in your response all the elements listed under the Requirements section. Prompts in the Inspiration section are not required; however, they may help you to fully think through your response.

Remember to review the Touchstone page for entry requirements, examples, and grading specifics.

**Name: Anthony Mascari**

**Date: 08/13/2023**

**Final Replit Program Share Link:** <https://github.com/Nice-Take/sophia_python.git>

**Main = pet\_selector.py**

## PART 1: Defining Your Problem

|  |
| --- |
| **Task**  State the problem you are planning to solve.  **Requirements**   * Describe the problem you are trying to solve for. * Describe any input data you expect to use. * Describe what the program will do to solve the problem. * Describe any outputs or results the program will provide.   **Inspiration**  When writing your entry below ask yourself the following questions:   * Why do you want to solve this particular problem? * What source(s) of data do you believe you will need? Will the user need to supply that data, or will you get it from an external file or another source? * Will you need to interact with the user throughout the program? Will users continually need to enter data in and see something to continue? * What are your expected results or what will be the end product? What will you need to tell a user of your program when it is complete? |
| **Problem:**  When it comes time for a family to choose a house pet, things can get complicated quickly. Tensions can rise and conflict can cloud the decision-making process even further. Often there is a need to organize thoughts, qualify and quantify numerous factors. These factors can become exceedingly difficult to keep track of mentally when emotions and tensions are high so a program to help a family solve this issue would be incredibly helpful.  We will need the user to input information that pertains to their particular circumstances. There will be initial user input then possibly a second round of required user input in order to provide the best recommendation possible. At the end of the program the user can expect to receive a single recommendation of what pet they should consider for their family.  **Required Input:**  We will need several entries from the user in order to help them best choose a pet.   1. How many people are in the family? 2. How much weight does that persons’ preference carry? 3. Does this person prefer a mammal or reptile pet? 4. Does this person prefer to cuddle or spectate? 5. Is anyone in the family allergic to pet dander? 6. Do you have noise restrictions where you live? 7. Annual pet budget? 8. How many days will you be away from your pet at a time and how often per year? 9. Do you have a yard?   **Program Solution:**  The program will operate by having a number of different pets hardcoded with appropriate attributes in a module called “pet\_options.py”. It will then take input for each member of the family, account for that user’s decision power and match the family with best fit pet.  **Desired Output:**  The program will provide the best fit pet solution for the current family and their set of circumstances. |

## PART 2: Working Through Specific Examples

|  |
| --- |
| **Task**  Write down clear and specific steps to solve a simple version of your problem you identified in Part 1.  **Requirements**  Complete the three steps below **for at least two distinct examples/scenarios**.   * State any necessary input data for your simplified problem. * Write clear and specific steps in English (not Python) detailing what the program will do to solve the problem. * Describe the specific result of your example/scenario.   **Inspiration**  When writing your entry below ask yourself the following questions:   * Are there any steps that you don’t fully understand? These are places to spend more time working out the details. Consider adding additional smaller steps in these spots. * Remember that a computer program is very literal. Are there any steps that are unclear? Try giving the steps of your example/scenario to a friend or family member to read through and ask you questions about parts they don’t understand. Rewrite these parts as clearly as you can. * Are there interesting edge cases for your program? Try to start one of your examples/scenarios with input that matches this edge case. How does it change how your program might work? |
| **Task:**  The first thing to appear in the program will be a welcome message and a brief explanation of what the program does, and what the user can expect to receive at the end of the program.  The user will then be prompted to answer a number of questions.  **Required Input:**   1. How many people are in the family?   *For each family member they will be asked:*   1. How much weight does this persons’ preference carry? 2. Does this person prefer a mammal or reptile pet? 3. Does this person prefer to cuddle or spectate?   *The remaining questions will only be asked of the user one time.*   1. Is anyone in the family allergic to pet dander? 2. Do you have noise restrictions where you live? 3. Annual pet budget? 4. How many days will you be away from your pet at a time and how often per year? 5. Do you have a yard?   **Steps:**   1. Receive input 2. Create objects containing information for each family/member 3. Compare the user input to the hardcoded “pet\_options.py” 4. Numerically score matching options between the pet\_option classes and the user\_input 5. Return the pet option with the highest score to the user with a brief explanation   **Scenario 1:**  User Input:   1. How many people are in the family? **1** 2. How much weight does that persons’ preference carry (1-10)? **10** 3. Does this person prefer a mammal or reptile pet? **Mammal** 4. Does this person prefer to cuddle or spectate? **Cuddle** 5. Is anyone in the family allergic to pet dander? **No** 6. Do you have noise restrictions where you live? **No** 7. Annual pet budget? $**1000** 8. How many days will you be away from your pet at a time? **2** 9. Do you have a yard? **Yes**   RESULT: Dog because they like to cuddle, are not allergic to dander, have enough budget and a fitting lifestyle to comfortably own a dog as a house-pet.  **Scenario 1:**  User Input:   1. How many people are in the family? **4** 2. How much weight does PERSON1’s preference carry (1-10)? **5** 3. Does this person prefer a mammal or reptile pet? **Mammal** 4. Does this person prefer to cuddle or spectate? **Cuddle** 5. How much weight does PERSON2’s preference carry (1-10)? **5** 6. Does this person prefer a mammal or reptile pet? **Mammal** 7. Does this person prefer to cuddle or spectate? **Spectate** 8. How much weight does PERSON3’s preference carry (1-10)? **10** 9. Does this person prefer a mammal or reptile pet? **Mammal** 10. Does this person prefer to cuddle or spectate? **Spectate** 11. How much weight does PERSON4’s preference carry (1-10)? **10** 12. Does this person prefer a mammal or reptile pet? **Reptile** 13. Does this person prefer to cuddle or spectate? **Spectate** 14. Is anyone in the family allergic to pet dander? **No** 15. Do you have noise restrictions where you live? **No** 16. Annual pet budget? $**1000** 17. How many days will you be away from your pet at a time? **2** 18. Do you have a yard? **Yes**   RESULT: Cat because they like on average prefer to spectate, are not allergic to dander, have enough budget and a fitting lifestyle to comfortably own a cat as a house-pet. |
| **Edge Case Possibilities:**  We may encounter a situation where everything is weighted evenly with an even number of family member where we have two identical match scores. This is a scenario we will have to keep an eye on as we are solving and test for in order to ensure we don’t encounter an error at runtime or a recommendation that is not useful to the end-user. |
|  |

## PART 3: Generalizing Into Pseudocode

|  |
| --- |
| **Task**  Write out the general sequence your program will use, including all specific examples/scenarios you provided in Part 2.  **Requirements**   * Write pseudocode for the program in English but refer to Python program elements where they are appropriate. The pseudocode should represent the full functionality of the program, not just a simplified version. Pseudocode is broken down enough that the details of the program are no longer in any paragraph form. One statement per line is ideal.   **Help with writing pseudocode**   * Here are a few links that can help you write pseudocode with examples. Remember to check out part 3 of the Example Journal Template Submission if you have not already. Note: everyone will write pseudocode differently. There is no right or wrong way to write it other than to make sure you write it clearly and in as much detail as you can so that it should be easy to convert it to code later.   + <https://www.geeksforgeeks.org/how-to-write-a-pseudo-code/>   + <https://www.wikihow.com/Write-Pseudocode>   **Inspiration**  When writing your entry below ask yourself the following questions:   * Do you see common program elements and patterns in your specific examples/scenarios in Part 2, like variables, conditionals, functions, loops, and classes? These should be part of your pseudocode for the general sequence as well. * Are there places where the steps for your examples/scenarios in Part 2 diverged? These may be places where errors may occur later in the project. Make note of them. * When you are finished with your pseudocode, does it make sense, even to a person that does not know Python? Aim for the clearest description of the steps, as this will make it easier to convert into program code later. |
| **Pseudocode:**  Welcome the user with a message to confirm they have successfully started the program.  Ask the user if they would like to engage in the decision making process?  If the user agrees, proceed to ask how many people are in the family.  Use that number of people to setup a list of people objects to store their corresponding information as the user answers the prompt questions.  Ask the preference questions for each family member until all objects in the previously generated list are filled out.  Use the weights and preferences of each family member to calculate the optimum choice for each category that requires input from each family member.  Create a single object that corresponds to the hard-coded values for the different potential-match animal types.  Compare the single family object to each potential-match animal and award points for each match.  Compare all animals and the animal with the highest score is the best fit for the family.  Present the user with the option that best-fits their family.  Present an end-message that notifies the user that the program has finished running. |
| **Another Example:**   1. How many people are in the family? **1** 2. How much weight does that persons’ preference carry (1-10)? **10** 3. Does this person prefer a mammal or reptile pet? **Mammal** 4. Does this person prefer to cuddle or spectate? **Cuddle** 5. Is anyone in the family allergic to pet dander? **No** 6. Do you have noise restrictions where you live? **No** 7. Annual pet budget? $**1000** 8. How many days will you be away from your pet at a time? **2** 9. Do you have a yard? **Yes** |

## PART 4: Testing Your Program

|  |
| --- |
| **Task**  While writing and testing your program code, describe your tests, record any errors, and state your approach to fixing the errors.  **Requirements**   * For at least one of your test cases, describe how your choices for the test helped you understand whether the program was running correctly or not.   For each error that occurs while writing and testing your code:   * Record the details of the error from Replit. A screenshot or copy-and-paste of the text into the journal entry is acceptable. * Describe what you attempted in order to fix the error. Clearly identify what approach was the one that worked.   **Inspiration**  When writing your entry below ask yourself the following questions:   * Have you tested edge cases and special cases for the inputs of your program code? Often these unexpected values can cause errors in the operation of your program. * Have you tested opportunities for user error? If a user is asked to provide an input, what happens when they give the wrong type of input, like a letter instead of a number, or vice versa? * Did the outcome look the way you expected? Was it formatted correctly? * Does your output align with the solution to the problem you coded for? |
| **Task:**  While writing and testing my code, I encountered a number of issues. Most of the issues were fairly straight forward but I did run into something that completely had me stuck. At one point I attempted to create a list of objects then iterate over that list to remove objects that did not meet specific criteria. There were eight objects and four of them had the attribute that should have caused them to be removed from the list but the iteration would stop after only two of the objects had been removed.  I will include a screenshot of part of the problem-solving process I went through in order to find what was happening. I even went so far as to match the memory addresses of the different objects. While the memory addresses matched on the four items and I could print all four matching items, I could only remove two items out of the eight when I tried to use the .remove(thing) method.    Ultimately after hours, I was unable to solve this issue and still use the .remove() method so I created an additional list and appended items that met the criteria to be removed. I then used that “to\_be\_removed” list to compare to the original list and create a third list of the items that I wanted. This work-around left me feeling dirty and inefficient.  I am still researching the reason for the failure when using the .remove() method on objects in a list.  Another error I encountered that was far easier to solve. I needed to validate the user input for each question that was asked and handle the error if the user entered erroneous information. Since the user is asked quite a few questions during this program, all information had to be validated and all bad input had to be handled. I chose to do this as the information enters the program so I could avoid checking for types at every function and only allow the proper type of information into the proper place. See screenshot on the next page of how the validation function and error handling was done.  An example of this can be seen here: |
|  |

## PART 5: Commenting Your Program

|  |
| --- |
| **Task**  Submit your full program code, including thorough comments describing what each portion of the program should do when working correctly.  **Requirements**   * The purpose of the program and each of its parts should be clear to a reader that does not know the Python programming language.   **Inspiration**  When writing your entry, you are encouraged to consider the following:   * Is each section or sub-section of your code commented to describe what the code is doing? * Give your code with comments to a friend or family member to review. Add additional comments to spots that confuse them to make it clearer. |
| **REPO:**  https://github.com/Nice-Take/sophia\_python.git  *Below is the copy/pasted code from my project. I would recommend viewing on git if possible.*  **Main: pet\_selector.py**  import pet\_options  import util  import family\_profile  import time  util.welcome()  if util.ready\_check():  # Init list to add family member objects to for later calculations  family\_members = []  # Create person object to store the preference of each memeber of the family from user input  # Each of these steps requires validation upon initial input so we don't encounter an error during calculation  for person in range(util.set\_family\_size()):  # Begin taking and validation of user input  choice\_weight = util.validate\_int(f"What is Person #{person+1}'s preference weight? (1-10): ")  mammal\_v\_reptile = util.validate\_exact\_str(f"Does Person #{person+1} prefer", "mammal", "reptile")  cuddle\_v\_spectate = util.validate\_exact\_str(f"Does Person #{person+1} prefer to", "cuddle", "spectate")  # Create an object for each set of input for each person to stay organized and tidy  person = family\_profile.Person(preference\_weight = choice\_weight,  mammal\_or\_reptile = mammal\_v\_reptile,  cuddle\_or\_spectate = cuddle\_v\_spectate)  # Add the new person object to a list for later calculations  family\_members.append(person)  # Calculating the family's weighted preference and unpacking the return list into (2) corresponding variables  fam\_mammal\_v\_reptile, fam\_cuddle\_v\_spectate = (util.calculate\_family\_preferences(family\_members))    # Requesting the remaining necessary family situation information, the per-person input is compelete at this point  dander\_allergy = util.validate\_y\_n("Is anyone in the family allergic to pet dander? (Y/N): ")  noise\_restrictions = util.validate\_y\_n("Are there noise restrictions where you live? (Y/N): ")  yard = util.validate\_y\_n("Does your family have a yard? (Y/N): ")  budget = util.validate\_int("What is your annual pet budget? (USD): ")  days\_absent = util.validate\_int("How many days at a time will you be away?: ")  # Create family\_profile.Family object from the obtained information  fam\_profile = family\_profile.Family(family\_size=len(family\_members),  dander\_allergies = dander\_allergy,  noise\_restriction = noise\_restrictions,  annual\_budget = budget,  consecutive\_days\_away = days\_absent,  has\_yard = yard,  mammal\_or\_reptile = fam\_mammal\_v\_reptile,  cuddle\_or\_spectate = fam\_cuddle\_v\_spectate)  # Matching the best fit pet to the family's specifications  selected\_pet = util.match\_family\_to\_pet(fam\_profile, pet\_options.all\_possible\_pets)  # Getting the name of the selected pet for presentation to the user  new\_pet = selected\_pet.name  # Present the user with the result with timing to help build excitement  print(f"Thank you for using the Pet Selector!\nSit tight while we calculte your pet...")  time.sleep(1)  print(f"Hmmmm.....")  time.sleep(.75)  print("I've got it!")  time.sleep(.5)  print(f"\nThe best pet for your family is a...")  time.sleep(.75)  print(f"\n ----- [{new\_pet.capitalize()}] ----- \n")  time.sleep(2)  print("""Thank you and we hope we have helped  bring clarity to your decision!\n  ----- Bye! -----\n""")  **Module: pet\_options.py**  import typing  class Animal:  def \_\_init\_\_(self, name: str, mammal\_or\_reptile: str,  cuddle\_or\_spectate: str, has\_dander: bool,  has\_noise: bool, required\_budget: int,  requires\_yard: bool, days\_ok\_alone: int,  score: int):  self.name = name  self.mammal\_or\_reptile = mammal\_or\_reptile  self.cuddle\_or\_spectate = cuddle\_or\_spectate  self.has\_dander = has\_dander  self.has\_noise = has\_noise  self.required\_budget = required\_budget  self.requires\_yard = requires\_yard  self.days\_ok\_alone = days\_ok\_alone  self.score = score  def reset\_score(self):  self.score = 0  def add\_score(self, amount: int):  self.score += amount    def subtract\_score(self, amount: int):  self.score -= amount  dog = Animal(name = 'dog',  mammal\_or\_reptile = 'mammal',  cuddle\_or\_spectate = 'cuddle',  has\_dander = True,  has\_noise = True,  required\_budget = 500,  requires\_yard = True,  days\_ok\_alone = 1,  score = 0)  cat = Animal(name = 'cat',  mammal\_or\_reptile = 'mammal',  cuddle\_or\_spectate = 'spectate',  has\_dander = True,  has\_noise = False,  required\_budget = 1000,  requires\_yard = False,  days\_ok\_alone = 3,  score = 0)  hamster = Animal(name = 'hamster',  mammal\_or\_reptile = 'mammal',  cuddle\_or\_spectate = 'spectate',  has\_dander = True,  has\_noise = False,  required\_budget = 100,  requires\_yard = False,  days\_ok\_alone = 4,  score = 0)  bird = Animal(name = 'bird',  mammal\_or\_reptile = 'mammal',  cuddle\_or\_spectate = 'spectate',  has\_dander = True,  has\_noise = True,  required\_budget = 100,  requires\_yard = False,  days\_ok\_alone = 1,  score = 0)  gecko = Animal(name = 'gecko',  mammal\_or\_reptile = 'reptile',  cuddle\_or\_spectate = 'spectate',  has\_dander = False,  has\_noise = False,  required\_budget = 300,  requires\_yard = False,  days\_ok\_alone = 2,  score = 0)  bearded\_dragon = Animal(name = 'bearded dragon',  mammal\_or\_reptile = 'reptile',  cuddle\_or\_spectate = 'cuddle',  has\_dander = False,  has\_noise = False,  required\_budget = 600,  requires\_yard = False,  days\_ok\_alone = 3,  score = 0)  ball\_python = Animal(name = 'ball python',  mammal\_or\_reptile = 'reptile',  cuddle\_or\_spectate = 'spectate',  has\_dander = False,  has\_noise = False,  required\_budget = 1000,  requires\_yard = False,  days\_ok\_alone = 10,  score = 0)    chameleon = Animal(name = 'chameleon',  mammal\_or\_reptile = 'reptile',  cuddle\_or\_spectate = 'spectate',  has\_dander = False,  has\_noise = False,  required\_budget = 800,  requires\_yard = False,  days\_ok\_alone = 5,  score = 0)  all\_possible\_pets = [dog, cat, hamster, bird, gecko, bearded\_dragon, ball\_python, chameleon]  **Module: family\_profile.py**  import typing  class Person:  def \_\_init\_\_(self, preference\_weight: int,  mammal\_or\_reptile: str,  cuddle\_or\_spectate: str):  self.preference\_weight = preference\_weight  self.mammal\_or\_reptile = mammal\_or\_reptile  self.cuddle\_or\_spectate = cuddle\_or\_spectate  class Family:  def \_\_init\_\_(self, family\_size: int,  dander\_allergies: bool,  noise\_restriction: bool,  annual\_budget: int,  consecutive\_days\_away: int,  has\_yard: bool,  mammal\_or\_reptile: str,  cuddle\_or\_spectate: str):  self.family\_size = family\_size  self.dander\_allergies = dander\_allergies  self.noise\_restriction = noise\_restriction  self.annual\_budget = annual\_budget  self.consecutive\_days\_away = consecutive\_days\_away  self.has\_yard = has\_yard  self.mammal\_or\_reptile = mammal\_or\_reptile  self.cuddle\_or\_spectate = cuddle\_or\_spectate  def print\_all\_attribs(self):  print("\n\n ----- [FAMILY SUMMARY] -----\n")  print(f"Family Size: {self.family\_size}")  print(f"Dander Allergy: {self.dander\_allergies}")  print(f"Noise Regulations: {self.noise\_restriction}")  print(f"Annual Budget: {self.annual\_budget}")  print(f"Days Away: {self.consecutive\_days\_away}")  print(f"Has Yard: {self.has\_yard}")  print(f"Cuddle/Spectate: {self.cuddle\_or\_spectate}")  print(f"Mammal/Reptile: {self.mammal\_or\_reptile}")  **Module: util.py**  import time  import typing  import family\_profile  import pet\_options  def welcome() -> None:  """  Prints a welcome message notifying the user that the  program has begun. Also prints a description of the  expected process and what the result will be.  """  welcome\_msg = "Hello and welcome to the Pet Selector!"  program\_brief\_msg = "Answer a few questions to find your best fit pet!"  print(welcome\_msg)  time.sleep(1)  print(program\_brief\_msg)  time.sleep(1)  def ready\_check() -> bool:  """  Verifying that the user is ready to run the program.  Anything other than input 'Y' or 'y' will return False.  """  begin = input("Are you ready to begin? (Y/N): ")  if begin.upper() == 'Y':  print("Here we go!\n")  return True  else:  print("You have indicated you are not ready to proceed.\nProgram Ended.")  return False  def set\_family\_size() -> int:  """  Retrieves and validates the user's input family size as an int.  If anything other than an integer is returned, the user will  be asked to try again until an integer is successfully entered.  """  validating = True  while validating == True:  family\_size = input("How many people are in your same-house family?: ")  # Handling any accidental spaces entered  family\_size.replace(" ", "")  # Validating input as numeric  if family\_size.isdigit():  validated\_family\_size = int(family\_size)  # Input has been validated, end loop and return  validating = False  else:  # Provide error message to the user and reattempt  print("Oops!\nPlease enter a valid whole number. ")  return validated\_family\_size  def calculate\_family\_preferences(family\_members: list) -> list:  """  Takes in a list of family\_profile.Person objects, then calculates  the and returns a list of strings containing the corresponding  value with the highest weighted matching score. This result will  set the family's preference for - mammal vs reptile - & - cuddle vs  spectate.  Mammal vs reptile result is item at index 0  Cuddle vs spectate result is item at index 1  """  # Init neutral scores for decision categoires  mammal\_score = 0  reptile\_score = 0  cuddle\_score = 0  spectate\_score = 0  # Assigning points for mammal/reptile & cuddle/spectate using weighted user input values  # To determine the families overall weighted preference  for member in family\_members:  weight\_multiplier = member.preference\_weight  if member.mammal\_or\_reptile == 'mammal':  mammal\_score += 10 \* weight\_multiplier  else:  reptile\_score += 10 \* weight\_multiplier    if member.cuddle\_or\_spectate == 'cuddle':  cuddle\_score += 10 \* weight\_multiplier  else:  spectate\_score += 10 \* weight\_multiplier  # Compare scores and set result for mammal/reptile  if mammal\_score >= reptile\_score:  mammal\_vs\_reptile\_choice = 'mammal'  else:  mammal\_vs\_reptile\_choice = 'reptile'  # Compare scores and set result for cuddle/spectate  if cuddle\_score >= spectate\_score:  cuddle\_vs\_spectate\_choice = 'cuddle'  else:  cuddle\_vs\_spectate\_choice = 'spectate'  family\_preferences = [mammal\_vs\_reptile\_choice, cuddle\_vs\_spectate\_choice]  return family\_preferences  def validate\_y\_n(prompt: str) -> bool:  """  This function wraps the standard input() with measures to validate the user's  entry and convert it to a boolean.  """  raw\_input = input(prompt)  # Removing accidental space from input  raw\_input.replace(" ", "")  validating = True  validated\_output = bool  while validating:  if raw\_input.lower() == 'y':  validated\_output = True  validating = False  elif raw\_input.lower() == 'n':  validated\_output = False  validating = False  else:  raw\_input = input("Enter 'Y' for YES or 'N' for NO: ")  return validated\_output  def validate\_int(prompt: str) -> int:  """  This function wraps the standard input() with validation checks.  """  raw\_input = clean\_str\_for\_int\_conversion(input(prompt))  validating = True  validated\_output = int  while validating:  try:  validated\_output = int(raw\_input)  validating = False  except:  raw\_input = clean\_str\_for\_int\_conversion(input("Please enter only a valid number: "))  return validated\_output  def clean\_str\_for\_int\_conversion(input: str) -> str:  """  Cleans the string parameter input of symbols commonly entered by users for USD value.  Examples: '$', '.', ',', ' ' |  If a '.' is removed it was likely intended as a decimal and has meaning  so we round the int instead of replace '.'  """  # Removing space from int input  cleaned = input.replace(" ", "")  # Removing dollar-sign from user input  cleaned = cleaned.replace("$", "")  # Removing commas from user input  cleaned = cleaned.replace(",", "")  # Removing periods  try:  cleaned = int(round(cleaned))  cleaned = str(cleaned)  except:  cleaned = cleaned.replace(".", "")  return cleaned  def validate\_exact\_str(question\_phrase: str, string1: str, string2: str) -> str:  """  Takes 3 parameters. First is the phrase at the beginning of the prompt.  Next is the first string option 'string1' and third is the last 'string2'  option that the function user would like input from. The case entered  is not relevant as this function forces everything to lower before comparing.  """  raw\_input = input(f"{question\_phrase} ({string1} or {string2})?: ")  validated\_output = str  validating = True  while validating:  # Matching input to valid results and returning only once matched  if raw\_input.lower() == string1.lower():  validated\_output = string1.lower()  validating = False  elif raw\_input.lower() == string2.lower():  validated\_output = string2.lower()  validating = False  else:  raw\_input = input(f"Please reply with only '{string1}' or '{string2}: ")  return validated\_output    def match\_family\_to\_pet(family: family\_profile.Family, possible\_pets: list[pet\_options.Animal]) -> pet\_options.Animal:  """  Takes in a family object and a list of possible pets as pet\_options.Animal objects  and returns a pet\_options.Animal object that best-fits the family\_profile.Family.  Matches are calculated by matching values and assigning points for each match.  The Animal that has the highest match value score total will be the Animal  returned from this function.  """  remove\_pet\_list = [] # This is implemented because the cleaner .remove() was encoutering an error where only 2 items would remove from list  # Eliminate pets not categorically viable, ie. allergies, yard, noise restrictions  # These are done outside of the scoring loop to avoid unnecessary iterations through  # Animals that are not a viable match for the family in any circumstance.  # First we check for allergies as this is the single factor that will exclude the most pets  if family.dander\_allergies:  for pet in possible\_pets:  if pet.has\_dander:  remove\_pet\_list.append(pet)  # If the pet does not meet noise restriction requirements it is removed here  if family.noise\_restriction:  for pet in possible\_pets:  if pet.has\_noise and pet not in remove\_pet\_list:  remove\_pet\_list.append(pet)  # Here we remove any pet that does not meet the yard requirement  if family.has\_yard == False:  for pet in possible\_pets:  if pet.requires\_yard:  remove\_pet\_list.append(pet)  # Create narrowed list for scoring after eliminating animals that absolutely won't work with the family  viable\_pets = []  for pet in possible\_pets:  if pet in possible\_pets and pet not in remove\_pet\_list:  viable\_pets.append(pet)  # Begin scoring loop based upon remaining family and pet attributes  for pet in viable\_pets:  if pet.mammal\_or\_reptile == family.mammal\_or\_reptile:  pet.add\_score(10)  if pet.cuddle\_or\_spectate == family.cuddle\_or\_spectate:  pet.add\_score(10)  if pet.required\_budget <= family.annual\_budget:  pet.add\_score(8)  if pet.required\_budget >= family.annual\_budget:  pet.subtract\_score(-2)  if pet.days\_ok\_alone >= family.consecutive\_days\_away:  pet.add\_score(2)  else:  pet.subtract\_score(3)  # Retrieve the pet with the highest score  best\_fit\_pet = pet\_options.Animal  highscore = 0  for pet in viable\_pets:  if pet.score > highscore:  highscore = pet.score  best\_fit\_pet = pet  # Returing the pet with the highest overall score  return best\_fit\_pet |

## PART 6: Your Completed Program

|  |
| --- |
| **Task**  Provide the Replit link to your full program code.  **Requirements**   * The program must work correctly with all the comments included in the program.   **Inspiration**   * Check before submitting your touchstone that your final version of the program is running successfully. |
| https://github.com/Nice-Take/sophia\_python.git |