# **CIS 61 :: Lab 05 - List and Data Abstraction**

#### **Student Name: Jonathan Adrianto Saleh**

NOTE: DO NOT USE LIST FUNCTIONS IN YOUR SOLUTIONS.

Part 1: Lists

### Q1: If This Not That

#### Define if\_this\_not\_that, which takes a list of integers i\_list and an integer this. For each element in i\_list, if the element is larger than this, then print the element. Otherwise, print "that".

|  |
| --- |
| def if\_this\_not\_that(i\_list, this):  """Define a function which takes a list of integers `i\_list` and an integer `this`.  For each element in `i\_list`, print the element if it is larger than `this`;  otherwise, print the word "that".  >>> original\_list = [1, 2, 3, 4, 5]  >>> if\_this\_not\_that(original\_list, 3)  that  that  that  4  5  """  "\*\*\* YOUR CODE HERE \*\*\*" |

#### 

### Q2: Couple

Implement the function couple, which takes in two lists and returns a list that contains lists with i-th elements of two sequences coupled together. You can assume the lengths of two sequences are the same. Try using a list comprehension.

*Hint*: You may find the built in [rasnge](https://www.w3schools.com/python/ref_func_range.asp) function helpful.

|  |
| --- |
| **def** **couple**(s1, s2):  """Return a list that contains lists with i-th elements of two sequences  coupled together.  >>> s1 = [1, 2, 3]  >>> s2 = [4, 5, 6]  >>> couple(s1, s2)  [[1, 4], [2, 5], [3, 6]]  >>> s3 = ['c', 6]  >>> s4 = ['s', '1']  >>> couple(s3, s4)  [['c', 's'], [6, '1']]  """  **assert** len(s1) == len(s2)  "\*\*\* YOUR CODE HERE \*\*\*" |

### Q3: Enumerate

### Implement enumerate, which pairs the elements of a sequence with their indices, offset by a starting value. enumerate takes a sequence s and a starting value start. It returns a list of pairs, in which the i-th element is i + start paired with the i-th element of s. For example:

### >>> enumerate(['maps', 21, 47], start=1)

### >>> [[1, 'maps'], [2, 21], [3, 47]]

|  |
| --- |
| **def** **enumerate**(s, start=0):  """Returns a list of lists, where the i-th list contains i+start and  the i-th element of s.  >>> enumerate([6, 1, 'a'])  [[0, 6], [1, 1], [2, 'a']]  >>> enumerate('five', 5)  [[5, 'f'], [6, 'i'], [7, 'v'], [8, 'e']]  """  "\*\*\* YOUR CODE HERE \*\*\*" |

### Q4: Squares only

Implement the function squares, which takes in a list of positive integers. It returns a list that contains the square roots of the elements of the original list that are perfect squares. Try using a list comprehension.

You may find the round function useful.

>>> round(10.5)

10

>>> round(10.51)

11

|  |
| --- |
| **def** **squares**(s):  """Returns a new list containing square roots of the elements of the original list  that are perfect squares.  >>> seq = [8, 49, 8, 9, 2, 1, 100, 102]  >>> squares(seq)  [7, 3, 1, 10]  >>> seq = [500, 30]  >>> squares(seq)  []  """  "\*\*\* YOUR CODE HERE \*\*\*"  >>> def squares(s):  return[sqrt(s[i]) for i in range(len(s)) if round(sqrt(s[i])) == sqrt(s[i])] |

### Q5: Key of Min Value

The built-in min function takes a collection of elements (such as a list or a dictionary) and returns the collection's smallest element. The min function also takes in an optional keyword argument called key, which must be a one-argument function. The key function is called on each element of the collection, and the return values are used for comparison. For example:

>>> min([-1, 0, 1]) # no key argument; return smallest input

-1

>>> min([-1, 0, 1], key=lambda x: x\*x) # return input with the smallest square

0

Implement key\_of\_min\_value, which takes in a dictionary d and returns the key that corresponds to the minimum value in d. This behavior differs from just calling min on a dictionary, which would return the smallest key. *Make sure your solution is only one line and uses the min function.*

|  |
| --- |
| **def** **key\_of\_min\_value**(d):  """Returns the key in a dict d that corresponds to the minimum value of d.  >>> letters = {'a': 6, 'b': 5, 'c': 4, 'd': 5}  >>> min(letters)  'a'  >>> key\_of\_min\_value(letters)  'c'  """  "\*\*\* YOUR CODE HERE \*\*\*" |

### Q6: Flatten

Write a function flatten that takes a (possibly deep) list and "flattens" it. For example:

>>> lst = [1, [[2], 3], 4, [5, 6]]

>>> flatten(lst)

[1, 2, 3, 4, 5, 6]

*Hint*: you can check if something is a list by using the built-in type function. For example,

>>> type(3) == list

False

>>> type([1, 2, 3]) == list

True

*Hint 2*: you can use recursion

|  |
| --- |
| **def** **flatten**(lst):  """Returns a flattened version of lst.  >>> flatten([1, 2, 3]) # normal list  [1, 2, 3]  >>> x = [1, [2, 3], 4] # deep list  >>> flatten(x)  [1, 2, 3, 4]  >>> x = [[1, [1, 1]], 1, [1, 1]] # deep list  >>> flatten(x)  [1, 1, 1, 1, 1, 1]  """  "\*\*\* YOUR CODE HERE \*\*\*" |

### Q7: Add Characters (Optional)

Given two words, w1 and w2, we say w1 is a subsequence of w2 if all the letters in w1 appear in w2 in the same order (but not necessarily all together). That is, you can add letters to any position in w1 to get w2. For example, "sing" is a substring of "ab**s**orb**ing**" and "cat" is a substring of "**c**ontr**a**s**t**".

Implement add\_chars, which takes in w1 and w2, where w1 is a substring of w2. It should return a string containing the characters you need to add to w1 to get w2. **Your solution must use recursion**.

In the example above, you need to add the characters "aborb" to "sing" to get "absorbing", and you need to add "ontrs" to "cat" to get "contrast".

The letters in the string you return should be in the order you have to add them from left to right. If there are multiple characters in the w2 that could correspond to characters in w1, use the leftmost one. For example, add\_words("coy", "cacophony") should return "acphon", not "caphon" because the first "c" in "coy" corresponds to the first "c" in "**c**ac**o**phon**y**".

|  |
| --- |
| def **add\_chars**(w1, w2):  """  Return a **string** containing the characters you need to **add** to w1 to **get** w2.  You may assume that w1 **is** a subsequence of w2.  >>> **add\_chars**("owl", "howl")  'h'  >>> **add\_chars**("want", "wanton")  '**on**'  >>> **add\_chars**("rat", "radiate")  'diae'  >>> **add\_chars**("a", "prepare")  'prepre'  >>> **add\_chars**("resin", "recursion")  'curo'  >>> **add\_chars**("fin", "effusion")  'efuso'  >>> **add\_chars**("coy", "cacophony")  'acphon'  """  "\*\*\* YOUR CODE HERE \*\*\*" |

Part 2: Data Abstraction

### City Data Abstraction

Say we have an abstract data type for cities. A city has a name, a latitude coordinate, and a longitude coordinate.

|  |
| --- |
| **def** make\_city(name, lat, lon):  """  >>> city = make\_city('Berkeley', 0, 1)  >>> get\_name(city)  'Berkeley'  >>> get\_lat(city)  0  >>> get\_lon(city)  1  """  **return** [name, lat, lon]  **def** get\_name(city):  """  >>> city = make\_city('Berkeley', 0, 1)  >>> get\_name(city)  'Berkeley'  """  **return** city[0]  **def** get\_lat(city):  """  >>> city = make\_city('Berkeley', 0, 1)  >>> get\_lat(city)  0  """  **return** city[1]  **def** get\_lon(city):  """  >>> city = make\_city('Berkeley', 0, 1)  >>> get\_lon(city)  1  """  **return** city[2] |

Our ADT has one **constructor**:

* make\_city(name, lat, lon): Creates a city object with the given name, latitude, and longitude.

We also have the following **selectors** in order to get the information for each city:

* get\_name(city): Returns the city's name
* get\_lat(city): Returns the city's latitude
* get\_lon(city): Returns the city's longitude

Here is how we would use the constructor and selectors to create cities and extract their information:

|  |
| --- |
| >>> berkeley = make\_city('Berkeley', 122, 37) >>> get\_name(berkeley) 'Berkeley'  >>> get\_lat(berkeley) 122  >>> new\_york = make\_city('New York City', 74, 40)  >>> get\_lon(new\_york) 40 |

### Q8: Distance

We will now implement the function distance, which computes the distance between two city objects. Recall that the distance between two coordinate pairs (x1, y1) and (x2, y2) can be found by calculating the sqrt of (x1 - x2)\*\*2 + (y1 - y2)\*\*2. We have already imported sqrt for your convenience. Use the latitude and longitude of a city as its coordinates; you'll need to use the selectors to access this info!

|  |
| --- |
| **from math import sqrt**  **def distance(city1, city2):**  **"""**  **>>> city1 = make\_city('city1', 0, 1)**  **>>> city2 = make\_city('city2', 0, 2)**  **>>> distance(city1, city2)**  **1.0**  **>>> city3 = make\_city('city3', 6.5, 12)**  **>>> city4 = make\_city('city4', 2.5, 15)**  **>>> distance(city3, city4)**  **5.0**  **"""**  **"\*\*\* YOUR CODE HERE \*\*\*** |
|  |

### Q9: Closer city

Next, implement closer\_city, a function that takes a latitude, longitude, and two cities, and returns the name of the city that is relatively closer to the provided latitude and longitude.

You may only use the selectors and constructors introduced above and the distance function you just defined for this question.

**Hint**: How can use your distance function to find the distance between the given location and each of the given cities?

|  |
| --- |
| **def** **closer\_city**(lat, lon, city1, city2):  """  Returns the name of either city1 or city2, whichever is closest to  coordinate (lat, lon).  >>> berkeley = make\_city('Berkeley', 37.87, 112.26)  >>> stanford = make\_city('Stanford', 34.05, 118.25)  >>> closer\_city(38.33, 121.44, berkeley, stanford)  'Stanford'  >>> bucharest = make\_city('Bucharest', 44.43, 26.10)  >>> vienna = make\_city('Vienna', 48.20, 16.37)  >>> closer\_city(41.29, 174.78, bucharest, vienna)  'Bucharest'  """  "\*\*\* YOUR CODE HERE \*\*\*" |