Discussion 4: Python Lists

This is an online worksheet that you can work on during discussions. Your work is not graded and you do not need to submit anything. The last section of most worksheets is Exam Prep, which will typically only be taught by your TA if you are in an Exam Prep section. You are of course more than welcome to work on Exam Prep problems on your own.

Lists

A *sequence* is an ordered collection of values. It has two fundamental properties: length and element selection. In this discussion, we'll explore one of Python's data types, the *list*, which implements this abstraction.

In Python, we can have lists of whatever values we want, be it numbers, strings, functions, or even other lists! Furthermore, the types of the list's contents need not be the same. In other words, the list need not be homogenous.

Lists can be created using square braces. Their elements can be accessed (or *indexed*) with square braces. Lists are zero-indexed: to access the first element, we must index at 0; to access the `i` th element, we must index at `i - 1`.

We can also index with negative numbers. These begin indexing at the end of the list, so the index `-1` is equivalent to the index `len(list) - 1` and index `-2` is the same as `len(list) - 2`.

Let's try out some indexing:

```
>>> fantasy_team = ['aaron rodgers', 'desean jackson']
>>> print(fantasy_team)
['aaron rodgers', 'desean jackson']
>>> fantasy_team[0]
'aaron rodgers'
>>> fantasy_team[len(fantasy_team) - 1]
'desean jackson'
>>> fantasy_team[-1]
'desean jackson'
```

List slicing

If we want to access more than one element of a list at a time, we can use a *slice*. Slicing a sequence is very similar to indexing. We specify a starting index and an ending index, separated by a colon. Python creates a new list with the elements from the starting index up to (but not including) the ending index.

We can also specify a step size, which tells Python how to collect values for us. For example, if we set step size to 2, the returned list will include every **other** value, from the starting index until the ending index. A negative step size indicates that we are stepping backwards through a list when collecting values.

You can also choose not to specify any/all of the slice arguments. Python will perform some default behaviour if this is the case:

- If the step size is left out, the default step size is 1.
- If the start index is left out, the default start index is the beginning of the list.
- If the end index is left out, the default end index is the end of the list.
- If the step size is negative, the default start index becomes the end of the list, and the default end index

becomes the beginning of the list.

Thus, `lst[:]` creates a list that is identical to `lst` (a copy of `lst`). `lst[::-1]` creates a list that has the same elements of `lst`, but reversed. Those rules still apply if more than just the step size is specified e.g. `lst[3::-1]`.

```
>>> directors = ['jenkins', 'spielberg', 'bigelow', 'kubrick']
>>> directors[:2]
['jenkins', 'spielberg']
>>> directors[1:3]
['spielberg', 'bigelow']
>>> directors[1:]
['spielberg', 'bigelow', 'kubrick']
>>> directors[0:4:2]
['jenkins', 'bigelow']
>>> directors[::-1]
['kubrick', 'bigelow', 'spielberg', 'jenkins']
```

List comprehensions

A **list comprehension** is a compact way to create a list whose elements are the results of applying a fixed expression to elements in another sequence.

```
[<map exp> for <name> in <iter exp> if <filter exp>]
```

It might be helpful to note that we can rewrite a list comprehension as an equivalent for statement. See the example to the right.

Let's break down an example:

```
[x * x - 3 for x in [1, 2, 3, 4, 5] if x % 2 == 1]
```

In this list comprehension, we are creating a new list after performing a series of operations to our initial sequence `[1, 2, 3, 4, 5]`. We only keep the elements that satisfy the filter expression `x \% 2 == 1` (`1`, `3`, and `5`). For each retained element, we apply the map expression `x*x - 3` before adding it to the new list that we are creating, resulting in the output `[-2, 6, 22]`.

> Note: The `if` clause in a list comprehension is optional.

Questions

Q1: WWPD: Lists

What would Python display?

```
>>> a = [1, 5, 4, [2, 3], 3]
>>> print(a[0], a[-1])
```

```
>>>len(a)
```

```
>>> 2 in a
```

```
>>> a[3][0]
```

Q2: Even weighted

Write a function that takes a list `s` and returns a new list that keeps only the even-indexed elements of `s` and multiplies them by their corresponding index.

```
def even_weighted(s):
    """
    >>> x = [1, 2, 3, 4, 5, 6]
    >>> even_weighted(x)
    [0, 6, 20]
    """
    return [_____________________________________________]


```

Q3: Closest Number

Write a function that takes in a list of numbers `nums` and a target number `target` and returns the number in `nums` that is the closest to `target`. If there's a tie, return the number that shows up earlier in the list. You should do this in one line.

Hint: To find how close two numbers are, you can use `abs(x - y)` **Hint 2:** Use the `min` function and pass in a `key` function.

```
def closest_number(nums, target):
    """
    >>> closest_number([1, 4, 5, 6, 7], 2)
    1
    >>> closest_number([3, 1, 5, 6, 13], 4) #  choose the earlier number since there's a tie
    3
    >>> closest_number([34, 102, 8, 5, 23], 25)
    23
    """


```

Q4: Max Product

Write a function that takes in a list and returns the maximum product that can be formed using nonconsecutive elements of the list. The input list will contain only numbers greater than or equal to 1.

```
def max_product(s):
    """Return the maximum product that can be formed using non-consecutive
    elements of s.

    >>> max_product([10,3,1,9,2]) # 10 * 9
    90
    >>> max_product([5,10,5,10,5]) # 5 * 5 * 5
    125
    >>> max_product([])
    1
    """


```

Dictionaries

Dictionaries are data structures which map keys to values. Dictionaries in Python are unordered, unlike real-world dictionaries — in other words, key-value pairs are not arranged in the dictionary in any particular order. Let's look at an example:

```
>>> pokemon = {'pikachu': 25, 'dragonair': 148, 'mew': 151}
>>> pokemon['pikachu']
25
>>> pokemon['jolteon'] = 135
>>> pokemon
{'jolteon': 135, 'pikachu': 25, 'dragonair': 148, 'mew': 151}
>>> pokemon['ditto'] = 25
>>> pokemon
{'jolteon': 135, 'pikachu': 25, 'dragonair': 148,
'ditto': 25, 'mew': 151}
```

The *keys* of a dictionary can be any *immutable* value, such as numbers, strings, and tuples.[1] Dictionaries themselves are mutable; we can add, remove, and change entries after creation. There is only one value per key, however — if we assign a new value to the same key, it overrides any previous value which might have existed.

To access the value of `dictionary` at `key`, use the syntax `dictionary[key]`.

Element selection and reassignment work similarly to sequences, except the square brackets contain the key, not an index.

[1] To be exact, keys must be *hashable*, which is out of scope for this course. This means that some mutable objects, such as classes, can be used as dictionary keys.

Questions

Q5: WWPD: Dictionaries

What would Python display? Assume the following code block has been run:

```
>>> pokemon = {'pikachu': 25, 'dragonair': 148}
```

```
>>> pokemon
```

```
>>> 'mewtwo' in pokemon
```

```
>>> len(pokemon)
```

```
>>> pokemon['mew'] = pokemon['pikachu']
>>> pokemon[25] = 'pikachu'
>>> pokemon
```

```
>>> pokemon['mewtwo'] = pokemon['mew'] * 2
>>> pokemon
```

```
>>> pokemon[['firetype', 'flying']] = 146
```

Note that the last example demonstrates that dictionaries cannot use other mutable data structures as keys. However, dictionaries can be arbitrarily deep, meaning the *values* of a dictionary can be themselves dictionaries.

Q6: Group By

Write a function that takes in a list `s` and a function `fn` and returns a dictionary.

The values of the dictionary are lists of elements from `s`. Each element `e` in a list should be constructed such that `fn(e)` is the same for all elements in that list. The key for each value should be `fn(e)`. For each element `e` in s, check the value that calling `fn(e)` returns, and add `e` to the corresponding group.

```
def group_by(s, fn):
    """
    >>> group_by([12, 23, 14, 45], lambda p: p // 10)
    {1: [12, 14], 2: [23], 4: [45]}
    >>> group_by(range(-3, 4), lambda x: x * x)
    {9: [-3, 3], 4: [-2, 2], 1: [-1, 1], 0: [0]}
    """
    grouped = {}
    "*** YOUR CODE HERE ***"
    for _____________________:
        "*** YOUR CODE HERE ***"
        key = ___________________
        "*** YOUR CODE HERE ***"
        if ________________________________________:
            "*** YOUR CODE HERE ***"
             ____________________________________________
        else:
            "*** YOUR CODE HERE ***"
            grouped[key] = ____________________________________
    "*** YOUR CODE HERE ***"
    return _____________________________________


```

Exam Prep

Questions

Q7: Subset Sum (from Su15 MT 1)

Implement the `subset_sum(target, lst)` function: given a target integer `target` and a list of integers `lst`, return `True` if it is possible to add together any of the integers in `lst` to get the `target`. For example, `subset_sum(10, [-1, 5, 4, 6])` will return `True` (either -1 + 5 + 6 = 10 or 4 + 6 = 10), while `subset_sum(4, [5, -2, 12])` will return `False`.

> **Note**: an integer may appear multiple times in lst (for example, `[2, 4, 2, 3]`). An integer in `lst` can only be used once (for example, `subset_sum(4, [2, 3])` is `False` because we can only use the 2 once).

```
def subset_sum(target, lst):
    """Returns True if it is possible to add some of the integers in lst
    to get target.

    >>> subset_sum(10, [-1, 5, 4, 6])
    True
    >>> subset_sum(4, [5, -2, 12])
    False
    >>> subset_sum(-3, [5, -2, 2, -2, 1])
    True
    >>> subset_sum(0, [-1, -3, 15])     # Sum up none of the numbers to get 0
    True
    """
    if ______________________________
        return True
    elif ______________________________
        return False
    else:
        a = ______________________________
        b = ______________________________
        return a or b

```

Q8: Intersection (from Su15 MT 1)

Implement `intersection(lst_of_lsts)`, which takes a list of lists and returns a list of distinct elements that appear in all the lists in `lst_of_lsts`. If no number appears in all of the lists, return the empty list. You may assume that `lst_of_lsts` contains at least one list.

> **Hint:** recall that you can check if an element is in a list with the in operator:

```
>>> x = [1 , 2 , 3 , 4]
>>> 4 in x
True
>>> 5 in x
False
>>>
```

```
def intersection(lst_of_lsts):
    """Returns a list of distinct elements that appear in every list in
    lst_of_lsts.

    >>> lsts1 = [[1, 2, 3], [1, 3, 5]]
    >>> intersection(lsts1)
    [1, 3]
    >>> lsts2 = [[1, 4, 2, 6], [7, 2, 4], [4, 4]]
    >>> intersection(lsts2)
    [4]
    >>> lsts3 = [[1, 2, 3], [4, 5], [7, 8, 9, 10]]
    >>> intersection(lsts3)         # No number appears in all lists
    []
    >>> lsts4 = [[3, 3], [1, 2, 3, 3], [3, 4, 3, 5]]
    >>> intersection(lsts4)         # Return list of distinct elements
    [3]
    """
    elements = []
    for ____________________________________:
        condition = ____________________
        for ____________________________________:
            if ____________________________________:
                condition = ____________________
        if condition:
            elements = ___________________________________
    return elements
```

Q9: Wordify (from Sp17 Mock Midterm 1)

Did you know that in Python, you can access elements in a string exactly like you access elements in a list? For example, if we were given the string, `Gibbes` :

(a) Assigning to the variable `g` , allows us to access the 'G' with `g[0]` . (b) Slicing and concatenation are valid: `g[3:] + 't'` evaluates to 'best'.

Write a function that follows the specs below by creating a list of words out of a string, where a word has no spaces (' '). DO NOT USE LEN. You may only use the lines provided. You may not use any Python built-in sorting functions.

```
def wordify(s):
    """ Takes a string s and divides it into a list of words. Assume that the last element of
    Duplicate words allowed.
    >>> wordify ('sum total of human knowledge ')
    ['sum', 'total', 'of', 'human', 'knowledge']
    >>> wordify ('one should never use exclamation points in writing! ')
    ['one', 'should', 'never', 'use', 'exclamation', 'points', 'in', 'writing!']
    """
    start, end, lst = ____________________, ____________________, ____________________
    for letter in _____________
        if letter != ' ':
            end = ________________________________________
        elif start == end :
            start, end = ____________________, ____________________
        else :
            lst +=  [_____________________________________]
            start, end = ____________________, ____________________
    return ____________________
```

Diagnostic Review

If you'd like to review the diagnostic in your section, the PDF can be found here (/~cs61a/su21/exam/su21/diagnostic/61a-su21-diagnostic.pdf)