Sequences

Sequences are ordered collections of values that support element-selection and have length. We've worked with lists, but other Python types are also sequences, including strings.

Q1: Map, Filter, Reduce

Many languages provide map, filter, reduce functions for sequences. Python also provides these functions (and we'll formally introduce them later on in the course), but to help you better understand how they work, you'll be implementing these functions in the following problems.

> In Python, the map and filter built-ins have slightly different behavior than the my_map and my_filter functions we are defining here.

my_map takes in a one argument function fn and a sequence seq and returns a list containing fn applied to each element in seq.

```python
def my_map(fn, seq):
    """Applies fn onto each element in seq and returns a list.
    >>> my_map(lambda x: x*x, [1, 2, 3])                          \
    [1, 4, 9]
    """
    "*** YOUR CODE HERE ***"
        S = [ ]
        i = 0
        while i < len (seq):
            S.append( fn (seq[i]) )
            i += 1
        return S



# You can use more space on the back if you want
```

my_filter takes in a predicate function pred and a sequence seq and returns a list containing all elements in seq for which pred returns True.

```python
def my_filter(pred, seq):
    """Keeps elements in seq only if they satisfy pred.
    >>> my_filter(lambda x: x % 2 == 0, [1, 2, 3, 4])  # new list
    has only even-valued elements
    [2, 4]
    """
    "*** YOUR CODE HERE ***"
```

```
res = []
    for e in seq:
        if pred(e):
            res += [e]
    return res
```

```python
# You can use more space on the back if you want
```

`my_reduce` takes in a two argument function `combiner` and a non-empty sequence `seq` and combines the elements in `seq` into one value using `combiner`.

```python
def my_reduce(combiner, seq):
    """Combines elements in seq using combiner.
    seq will have at least one element.
    >>> my_reduce(lambda x, y: x + y, [1, 2, 3, 4])  # 1 + 2 + 3 + 4
    10
    >>> my_reduce(lambda x, y: x * y, [1, 2, 3, 4])  # 1 * 2 * 3 * 4
    24
    >>> my_reduce(lambda x, y: x * y, [4])
    4
    >>> my_reduce(lambda x, y: x + 2 * y, [1, 2, 3]) # (1 + 2 * 2) +
     2 * 3
    11
    """
    "*** YOUR CODE HERE ***"
    i = 1
    res = seq[0]
    while i < len(seq):
        res = combiner(res, seq[i])
        i += 1
    return res


# You can use more space on the back if you want
```

Q2: Count palindromes

The Python library defines `filter`, `map`, and `reduce`, which operate on Python sequences. Devise a function that counts the number of palindromic words (those that read the same backwards as forwards) in a tuple of words using only `lambda`, basic operations on strings, the tuple constructor, conditional expressions, and the functions `filter`, `map`, and `reduce`. Specifically, do not use recursion or any kind of loop:

```python
def count_palindromes(L):
    """The number of palindromic words in the sequence of strings
    L (ignoring case).

    >>> count_palindromes(("Acme", "Madam", "Pivot", "Pip"))
    2
    """
    return __len__(list(filter(lambda x: x.lower() == x[::-1].lower(), L)))
```

Hint: The easiest way to get the reversed version of a string `s` is to use the Python slicing notation trick `s[::-1]`. Also, the function `lower`, when called on strings, converts all of the characters in the string to lowercase. For instance, if the variable `s` contains the string "PyThoN", the expression `s.lower()` evaluates to "python".

Lists

A list is a data structure that can store multiple elements. Each element can be of any type, even a list itself. We write a list as a comma-separated list of expressions in square brackets:

```
>>> list_of_ints = [1, 2, 3, 4]
>>> list_of_bools = [True, True, False, False]
>>> nested_lists = [1, [2, 3], [4, [5]]]
```

Each element in the list has an index, with the index of the first element starting at 0. We say that lists are therefore "zero-indexed."

With list indexing, we can specify the index of the element we want to retrieve. A negative index represents starting from the end of the list, so the negative index `-i` is equivalent to the positive index `len(lst)-i`.

```
>>> lst = [6, 5, 4, 3, 2, 1, 0]
>>> lst[0]
6
>>> lst[3]
3
>>> lst[-1] # Same as lst[6]
0
```

List slicing

To create a copy of part or all of a list, we can use list slicing. The syntax to slice a list `lst` is: `lst[<start index>:<end index>:<step size>]`.

This expression evaluates to a new list containing the elements of `lst`:

- Starting at and including the element at `<start index>`.
- Up to but not including the element at `<end index>`.
- With `<step size>` as the difference between indices of elements to include.

If the start, end, or step size are not explicitly specified, Python has default values for them. A negative step size indicates that we are stepping backwards through a list when including elements.

```
>>> lst[:3]    # Start index defaults to 0
[6, 5, 4]
>>> lst[3:]    # End index defaults to len(lst)
[3, 2, 1, 0]
>>> lst[::-1]    # Make a reversed copy of the entire list
[0, 1, 2, 3, 4, 5, 6]
>>> lst[::2]   # Skip every other; step size defaults to 1 otherwise
[6, 4, 2, 0]
```

never forget the square brackets

List comprehensions

List comprehensions are a compact and powerful way of creating new lists out of sequences. The general syntax for a list comprehension is the following:

condition

```
[<expression> for <element> in <sequence> if <conditional>]
```

operation

iterate all the elements

where the `if <conditional>` section is optional.

The syntax is designed to read like English: "Compute the expression for each element in the sequence (if the conditional is true for that element)."

```
>>> [i**2 for i in [1, 2, 3, 4] if i % 2 == 0]
[4, 16]
```

This list comprehension will:

- Compute the expression `i**2`
- For each element `i` in the sequence `[1, 2, 3, 4]`
- Where `i % 2 == 0` (`i` is an even number),

and then put the resulting values of the expressions into a new list.

In other words, this list comprehension will create a new list that contains the square of every even element of the original list `[1, 2, 3, 4]`.

We can also rewrite a list comprehension as an equivalent `for` statement, such as for the example above:

```
>>> lst = []
>>> for i in [1, 2, 3, 4]:
...     if i % 2 == 0:
...         lst = lst + [i**2]
>>> lst
[4, 16]
```

Q3: WWPD: Lists

What would Python display?

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

```
>>>len(a)
```
5

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

```
>>> a[3][0]
```
2

Q4: 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 [i * s[i] for i in range(len(s)) if i%2==0]
```

Q5: 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.

+ ¡ recursive question

```python
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
    """
    if len(s) == 0:
        return 1
    else:
        return max(S[0] * max_product(S[2:]), max_product(S[1:]))




# You can use more space on the back if you want
```

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:

```python
>>> 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

Note: This worksheet is a problem bank—most TAs will not cover all the problems in discussion section.

The key can't be lists.

==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.

Q6: WWPD: Dictionaries

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

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

```
>>> pokemon
```
{'pikachu':25, 'draganair':148}

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

```
>>> len(pokemon)
```
2

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

```
>>> pokemon['mewtwo'] = pokemon['mew'] * 2
>>> pokemon
```
{25:'pikachu', 'pikachu':25, 'dragonair':148, 'mew':25, 'mewtwo': 50}

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

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.