CS 61A Summer 2022

Sequences

Discussion 4: July 5, 2022 Solutions

##Walkthrough Videos

Feel free to try these problems on the worksheet in discussion or on your own, and then come back to reference these walkthrough videos as you study.

> To see these videos, you should be logged into your berkeley.edu email.
>
> **Note**: We unfortunately only have walkthrough videos for some of the problems on this worksheet this semester, please use the problem names to find the corresponding videos.

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`.

```
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]
    """
    result = []
    for elem in seq:
        result += [fn(elem)]
    return result
```

`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.

```
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]
    """
    result = []
    for elem in seq:
        if pred(elem):
            result += [elem]
    return result
```

`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`.

```
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
    """
    total = seq[0]
    for elem in seq[1:]:
        total = combiner(total, elem)
    return total
```

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:

```
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 s: s.lower() == s[::-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]
```

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:

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

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

Video walkthrough

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]
```

The key point to note is that instead of iterating over each element in the list, we must instead iterate over the indices of the list. Otherwise, there's no way to tell if we should keep a given element.

One way of solving these problems is to try and write your solution as a `for` loop first, and then transform it into a list comprehension. The `for` loop solution might look something like this:

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

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.

```
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 s == []:
        return 1
    else:
        return max(max_product(s[1:]), s[0] * max_product(s[2:]))
```

At each step, we choose if we want to include the current number in our product or not:

- If we include the current number, we cannot use the adjacent number.
- If we don't use the current number, we try the adjacent number (and obviously ignore the current number).

The recursive calls represent these two alternate realities. Finally, we pick the one that gives us the largest product.

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.

Q6: WWPD: Dictionaries

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

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

```
>>> pokemon
```

{‘pikachu’: 25, ‘dragonair’: 148}

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

False

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

2

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

{‘pikachu’: 25, ‘dragonair’: 148, ‘mew’: 25, 25: ‘pikachu’}

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

{‘pikachu’: 25, ‘dragonair’: 148, ‘mew’: 25, 25: ‘pikachu’, ‘mewtwo’: 50}

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

Error: unhashable type

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.