Lab 4: Recursion, Python Lists

lab04.zip (lab04.zip)

Due by 11:59pm on Wednesday, September 20.

Starter Files

Download lab04.zip (lab04.zip). Inside the archive, you will find starter files for the questions in this lab, along with a copy of the Ok (ok) autograder.

Topics

Consult this section if you need a refresher on the material for this lab. It's okay to skip directly to the questions and refer back here should you get stuck.

Recursion

Lists

Sequences

List Slicing

List Comprehensions

Required Questions

Getting Started Videos

Lists

Q1: WWPD: Lists

Use Ok to test your knowledge with the following "What Would Python Display?" questions:

```
python3 ok -q lists-wwpd -u
```

Predict what Python will display when you type the following into the interpreter. Then try it to check your answers.

```
>>> s = [7//3, 5, [4, 0, 1], 2]
>>> s[0]
______

>>> s[2]
______

>>> s[-1]
______

>>> len(s)
______

>>> 4 in s
______

>>> 4 in s[2]
______

>>> s + [3 + 2, 9]
______

>>> s[2] * 2
______

>>> x = [1, 2, 3, 4]
>>> x[1:3]
______

>>> x[:2]
______

>>> x[1:]
______

>>> x[-2:3]
______

>>> x[-2:4]
______

>>> x[0:4:2]
______

>>> x[::-1]
```

```
______
```

Sequences

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.

Q2: Map

`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`.

Use only a single line for the body of the function. (Hint: use a list comprehension.)

```
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]
    >>> my_map(lambda x: abs(x), [1, -1, 5, 3, 0])
    [1, 1, 5, 3, 0]
    >>> my_map(lambda x: print(x), ['cs61a', 'summer', '2023'])
    cs61a
    summer
    2023
    [None, None, None]
    """
    return ______
```

Use Ok to test your code:

```
python3 ok -q my_map
```

Use Ok to run the local syntax checker (which checks that you used only a single line for the body of the function):

```
python3 ok -q my_map_syntax_check
```

Q3: Filter

`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`. (A predicate function is a function that takes in an argument and returns either `True` or `False`.)

Use only a single line for the body of the function. (Hint: use a list comprehension.)

```
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 ele
    [2, 4]
    >>> my_filter(lambda x: (x + 5) % 3 == 0, [1, 2, 3, 4, 5])
    [1, 4]
    >>> my_filter(lambda x: print(x), [1, 2, 3, 4, 5])
    1
    2
    3
    4
    5
    []
    >>> my_filter(lambda x: max(5, x) == 5, [1, 2, 3, 4, 5, 6, 7])
    [1, 2, 3, 4, 5]
    """
    return ______
```

Use Ok to test your code:

```
python3 ok -q my_filter
```

Use Ok to run the local syntax checker (which checks that you used only a single line for the body of the function):

```
python3 ok -q my_filter_syntax_check
```

Q4: Reduce

`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
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q my_reduce
```

Recursion

Q5: Double Eights

Write a recursive function that takes in a number `n` and determines if the digits contain two adjacent `8` s. You can assume that `n` is at least a two-digit number. You may have already done this problem iteratively as an Extra Practice problem in Lab 1.

> **Hint:** Remember what tools you can use in order to isolate digits of a number. If you have trouble figuring out how to implement the recursion, try first finding an iterative solution, and think about how you might be able to turn any loops in recursion.

```
def double_eights(n):
    """ Returns whether or not n has two digits in row that
    are the number 8. Assume n has at least two digits in it.

    >>> double_eights(1288)
    True
    >>> double_eights(880)
    True
    >>> double_eights(538835)
    True
    >>> double_eights(284682)
    False
    >>> double_eights(588138)
    True
    >>> double_eights(78)
    False
    >>> from construct_check import check
    >>> # ban iteration
    >>> check(LAB_SOURCE_FILE, 'double_eights', ['While', 'For'])
    True
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q double_eights
```

Q6: Merge

Write a function `merge` that takes 2 *sorted* lists `lst1` and `lst2`, and returns a new list that contains all the elements in the two lists in sorted order.

Note: Solve this question using recursion, not iteration.

```
def merge(lst1, lst2):
    """Merges two sorted lists.

    >>> s1 = [1, 3, 5]
    >>> s2 = [2, 4, 6]
    >>> merge(s1, s2)
    [1, 2, 3, 4, 5, 6]
    >>> s1
    [1, 3, 5]
    >>> s2
    [2, 4, 6]
    >>> merge([], [2, 4, 6])
    [2, 4, 6]
    >>> merge([1, 2, 3], [])
    [1, 2, 3]
    >>> merge([5, 7], [2, 4, 6])
    [2, 4, 5, 6, 7]
    >>> merge([2, 3, 4], [2, 4, 6])
    [2, 2, 3, 4, 4, 6]
    >>> from construct_check import check
    >>> # ban iteration
    >>> check(LAB_SOURCE_FILE, 'merge', ['While', 'For'])
    True
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q merge
```

Check Your Score Locally

You can locally check your score on each question of this assignment by running

```
python3 ok --score
```

This does NOT submit the assignment! When you are satisfied with your score, submit the assignment to Gradescope to receive credit for it.

Submit

Make sure to submit this assignment by uploading any files you've edited **to the appropriate Gradescope assignment.** For a refresher on how to do this, refer to Lab 00 (https://cs61a.org/lab/lab00/#submit-with-gradescope).

Optional Questions

These questions are optional, but you must complete them in order to be checked off before the end of the lab period. They are also useful practice!

Lists

Q7: Summation

Write a recursive implementation of `summation`, which takes a positive integer `n` and a function `term`. It applies `term` to every number from `1` to `n` including `n` and returns the sum.

Important: Use recursion; the tests will fail if you use any loops (for, while).

```
def summation(n, term):
    """Return the sum of numbers 1 through n (including n) with term applied to each numbe
    Implement using recursion!

    >>> summation(5, lambda x: x * x * x) # 1^3 + 2^3 + 3^3 + 4^3 + 5^3
    225
    >>> summation(9, lambda x: x + 1) # 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10
    54
    >>> summation(5, lambda x: 2**x) # 2^1 + 2^2 + 2^3 + 2^4 + 2^5
    62
    >>> # Do not use while/for loops!
    >>> from construct_check import check
    >>> # ban iteration
    >>> check(LAB_SOURCE_FILE, 'summation',
    ...       ['While', 'For'])
    True
    """
    assert n >= 1
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q summation
```

Sequences

Q8: 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 ______
```

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".

Use Ok to test your code:

```
python3 ok -q count_palindromes
```