Lab 3 Solutions lab03.zip (lab03.zip)

Solution Files

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.

Required Questions

Lists

Important: For all WWPD questions, type `Function` if you believe the answer is `<function...>`, `Error` if it errors, and `Nothing` if nothing is displayed.

Q1: WWPD: Lists & Ranges

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 interactive 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[2] + [3 + 2]
______

>>> 5 in s[2]
______

>>> s[2] * 2
______

>>> list(range(3, 6))
______

>>> range(3, 6)
______

>>> r = range(3, 6)
>>> [r[0], r[2]]
______

>>> range(4)[-1]
______
```

Q2: Print If

Implement `print_if`, which takes a list `s` and a one-argument function `f`. It prints each element `x` of `s` for which `f(x)` returns a true value.

```
def print_if(s, f):
    """Print each element of s for which f returns a true value.

    >>> print_if([3, 4, 5, 6], lambda x: x > 4)
    5
    6
    >>> result = print_if([3, 4, 5, 6], lambda x: x % 2 == 0)
    4
    6
    >>> print(result)  # print_if should return None
    None
    """
    for x in s:
        if f(x):
            print(x)
```

Use Ok to test your code:

```
python3 ok -q print_if
```

Q3: Close

Implement `close`, which takes a list of numbers `s` and a non-negative integer `k`. It returns how many of the elements of `s` are within `k` of their index. That is, the absolute value of the difference between the element and its index is less than or equal to `k`.

> Remember that list is "zero-indexed"; the index of the first element is `0`.

```
def close(s, k):
    """Return how many elements of s that are within k of their index.

    >>> t = [6, 2, 4, 3, 5]
    >>> close(t, 0)  # Only 3 is equal to its index
    1
    >>> close(t, 1)  # 2, 3, and 5 are within 1 of their index
    3
    >>> close(t, 2)  # 2, 3, 4, and 5 are all within 2 of their index
    4
    >>> close(list(range(10)), 0)
    10
    """
    count = 0
    for i in range(len(s)):  # Use a range to loop over indices
        if abs(i - s[i]) <= k:
            count += 1
    return count
```

Use Ok to test your code:

```
python3 ok -q close
```

List Comprehensions

Important: For all WWPD questions, type `Function` if you believe the answer is `<function...>`, `Error` if it errors, and `Nothing` if nothing is displayed.

Q4: WWPD: List Comprehensions

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

```
python3 ok -q list-comprehensions-wwpd -u
```

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

```
>>> [2 * x for x in range(4)]
______

>>> [y for y in [6, 1, 6, 1] if y > 2]
______

>>> [[1] + s for s in [[4], [5, 6]]]
______

>>> [z + 1 for z in range(10) if z % 3 == 0]
______
```

Q5: Close List

Implement `close_list`, which takes a list of numbers `s` and a non-negative integer `k`. It returns a list of the elements of `s` that are within `k` of their index. That is, the absolute value of the difference between the element and its index is less than or equal to `k`.

```
def close_list(s, k):
    """Return a list of the elements of s that are within k of their index.

    >>> t = [6, 2, 4, 3, 5]
    >>> close_list(t, 0)  # Only 3 is equal to its index
    [3]
    >>> close_list(t, 1)  # 2, 3, and 5 are within 1 of their index
    [2, 3, 5]
    >>> close_list(t, 2)  # 2, 3, 4, and 5 are all within 2 of their index
    [2, 4, 3, 5]
    """
    return [s[i] for i in range(len(s)) if abs(i - s[i]) <= k]
```

Use Ok to test your code:

```
python3 ok -q close_list
```

Q6: 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. Use a list comprehension.

> To find if `x` is a perfect square, you can check if `sqrt(x)` equals `round(sqrt(x))`.

```
from math import sqrt

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)
    []
    """
    return [round(n ** 0.5) for n in s if n == round(n ** 0.5) ** 2]
```

It might be helpful to construct a skeleton list comprehension to begin with:

```
[round(sqrt(x)) for x in s if is_perfect_square(x)]
```

This is great, but it requires that we have an `is_perfect_square` function. How might we check if something is a perfect square?

- If the square root of a number is a whole number, then it is a perfect square. For example, `sqrt(61) = 7.81024...` (not a perfect square) and `sqrt(49) = 7` (perfect square).
- Once we obtain the square root of the number, we just need to check if something is a whole number. The `is_perfect_square` function might look like:

  ```
  def is_perfect_square(x):
      return is_whole(sqrt(x))
  ```

- One last piece of the puzzle: to check if a number is whole, we just need to see if it has a decimal or not. The way we've chosen to do it in the solution is to compare the original number to the round version (thus removing all decimals), but a technique employing floor division (`//`) or something else entirely could work too.

We've written all these helper functions to solve this problem, but they are actually all very short. Therefore, we can just copy the body of each into the original list comprehension, arriving at the solution we finally present.

Video walkthrough:

[YouTube link (https://youtu.be/YwLFB9paET0)](https://youtu.be/YwLFB9paET0)

Use Ok to test your code:

```
python3 ok -q squares
```

Recursion

Q7: Double Eights

Write a **recursive** function that takes in a positive integer `n` and determines if its digits contain two adjacent `8`s (that is, two `8`s right next to each other).u

> **Hint:** Start by coming up with a recursive plan: the digits of a number have double eights if either (think of something that is straightforward to check) or double eights appear in the rest of the digits.

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

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

    >>> double_eights(1288)
    True
    >>> double_eights(880)
    True
    >>> double_eights(538835)
    True
    >>> double_eights(284682)
    False
    >>> double_eights(588138)
    True
    >>> double_eights(78)
    False
    >>> # ban iteration
    >>> from construct_check import check
    >>> check(LAB_SOURCE_FILE, 'double_eights', ['While', 'For'])
    True
    """
    last, second_last = n % 10, n // 10 % 10
    if last == 8 and second_last == 8:
        return True
    elif n < 100:
        return False
    return double_eights(n // 10)

    # Alternate solution
    last, second_last = n % 10, n // 10 % 10
    if n < 10:
        return False
    return (last == 8 and second_last == 8) or double_eights(n // 10)

    # Alternate solution with helper function:
    def helper(num, prev_eight):
        if num == 0:
            return False
        if num % 10 == 8:
            if prev_eight:
                return True
            return helper(num // 10, True)
        return helper(num // 10, False)
    return helper(n, False)
```

Use Ok to test your code:

```
python3 ok -q double_eights
```

Q8: Making Onions

Write a function `make_onion` that takes in two one-argument functions, `f` and `g`. It returns a function that takes in three arguments: `x`, `y`, and `limit`. The returned function returns `True` if it is possible to reach `y` from `x` using up to `limit` calls to `f` and `g`, and `False` otherwise.

For example, if `f` adds 1 and `g` doubles, then it is possible to reach 25 from 5 in four calls: `f(g(g(f(5))))`.

```
def make_onion(f, g):
    """Return a function can_reach(x, y, limit) that returns
    whether some call expression containing only f, g, and x with
    up to limit calls will give the result y.

    >>> up = lambda x: x + 1
    >>> double = lambda y: y * 2
    >>> can_reach = make_onion(up, double)
    >>> can_reach(5, 25, 4)      # 25 = up(double(double(up(5))))
    True
    >>> can_reach(5, 25, 3)      # Not possible
    False
    >>> can_reach(1, 1, 0)      # 1 = 1
    True
    >>> add_ing = lambda x: x + "ing"
    >>> add_end = lambda y: y + "end"
    >>> can_reach_string = make_onion(add_ing, add_end)
    >>> can_reach_string("cry", "crying", 1)      # "crying" = add_ing("cry")
    True
    >>> can_reach_string("un", "unending", 3)     # "unending" = add_ing(add_end("un"))
    True
    >>> can_reach_string("peach", "folding", 4)   # Not possible
    False
    """
    def can_reach(x, y, limit):
        if limit < 0:
            return False
        elif x == y:
            return True
        else:
            return can_reach(f(x), y, limit - 1) or can_reach(g(x), y, limit - 1)
    return can_reach
```

Use Ok to test your code:

```
python3 ok -q make_onion
```

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 Assignment

If you are in a regular section of CS 61A, fill out this lab attendance and feedback form (https://forms.gle/dHxj8gttNWRY6Ptm9). (If you are in the mega section, you don't need to fill out the form.)

Then, submit this assignment by uploading any files you've edited **to the appropriate Gradescope assignment.** Lab 00 (../lab00/#submit-with-gradescope) has detailed instructions.