Lab 3: Recursion, Python Lists

lab03.zip (lab03.zip)

Due by 11:59pm on Wednesday, September 25.

Starter Files

Download lab03.zip (lab03.zip).

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.

Lists

Lists

A list is a data structure that can hold an ordered collection of items. These items, known as elements, can be of any data type, including numbers, strings, or even other lists. A comma-separated list of expressions in square brackets creates a list:

```
>>> list_of_values = [2, 1, 3, True, 3]
>>> nested_list = [2, [1, 3], [True, [3]]]
```

Each position in a list has an index, with the left-most element indexed `0`.

```
>>> list_of_values[0]
2
>>> nested_list[1]
[1, 3]
```

A negative index counts from the end, with the right-most element indexed `-1`.

```
>>> nested_list[-1]
[True, [3]]
```

Adding lists creates a longer list containing the elements of the added lists.

```
>>> [1, 2] + [3] + [4, 5]
[1, 2, 3, 4, 5]
```

List Comprehensions

List Comprehensions

A list comprehension describes the elements in a list and evaluates to a new list containing those elements.

There are two forms:

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

Here's an example that starts with `[1, 2, 3, 4]`, picks out the even elements `2` and `4` using `if i % 2 == 0`, then squares each of these using `i*i`. The purpose of `for i` is to give a name to each element in `[1, 2, 3, 4]`.

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

This list comprehension evaluates to a list of:

- The value of `i*i`
- For each element `i` in the sequence `[1, 2, 3, 4]`
- For which `i % 2 == 0`

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:

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

For Loops

For Statements

A `for` statement executes code for each element of a sequence, such as a list or range. Each time the code is executed, the name right after `for` is bound to a different element of the sequence.

```
for <name> in <expression>:
    <suite>
```

First, `<expression>` is evaluated. It must evaluate to a sequence. Then, for each element in the sequence in order,

1. `<name>` is bound to the element.
2. `<suite>` is executed.

Here is an example:

```
for x in [-1, 4, 2, 0, 5]:
    print("Current elem:", x)
```

This would display the following:

```
Current elem: -1
Current elem: 4
Current elem: 2
Current elem: 0
Current elem: 5
```

Ranges

Ranges

A range is a data structure that holds integer sequences. A range can be created by:

- `range(stop)` contains 0, 1, ..., `stop` - 1
- `range(start, stop)` contains `start`, `start` + 1, ..., `stop` - 1

Notice how the range function doesn't include the `stop` value; it generates numbers up to, but not including, the `stop` value.

For example:

```
>>> for i in range(3):
...     print(i)
...
0
1
2
```

While ranges and lists are both sequences (https://en.wikibooks.org/wiki/Python_Programming/Sequences), a range object is different from a list. A range can be converted to a list by calling `list()`:

```
>>> range(3, 6)
range(3, 6)  # this is a range object
>>> list(range(3, 6))
[3, 4, 5]  # list() converts the range object to a list
>>> list(range(5))
[0, 1, 2, 3, 4]
>>> list(range(1, 6))
[1, 2, 3, 4, 5]
```

Required Questions

Getting Started Videos

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

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
        "*** YOUR CODE HERE ***"
    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 [___ for i in range(len(s)) if ___]
```

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 [___ for n in s if ___]
```

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).

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

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 ____
        elif x == y:
            return ____
        else:
            return can_reach(____, ____, limit - 1) or can_reach(____, ____, 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.