Lab 5: Mutability, Orders of Growth

lab05.zip (lab05.zip)

Due by 11:59pm on Tuesday, July 13.

Starter Files

Download lab05.zip (lab05.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.

Mutability

Mutability

We say that an object is **mutable** if its state can change as code is executed. The process of changing an object's state is called **mutation**. Examples of mutable objects include lists and dictionaries. Examples of objects that are *not* mutable include tuples and functions.

We have seen how to use the `==` operator to check if two expressions evaluate to *equal* values. We now introduce a new comparison operator, `is`, that checks whether two expressions evaluate to the *same* values.

Wait, what's the difference? For primitive values, there is none:

```
>>> 2 + 2 == 3 + 1
True
>>> 2 + 2 is 3 + 1
True
```

This is because all primitives have the same *identity* under the hood. However, with non-primitive values, such as lists, each object has its own identity. That means you can construct two objects that may look exactly the same but have different identities.

```
>>> lst1 = [1, 2, 3, 4]
>>> lst2 = [1, 2, 3, 4]
>>> lst1 == lst2
True
>>> lst1 is lst2
False
```

Here, although the lists referred to by `lst1` and `lst2` have *equal* contents, they are not the *same* object. In other words, they are the same in terms of equality, but not in terms of identity.

This is important in our discussion of mutability because when we mutate an object, we simply change its state, *not* its identity.

```
>>> lst1 = [1, 2, 3, 4]
>>> lst2 = lst1
>>> lst1.append(5)
>>> lst2
[1, 2, 3, 4, 5]
>>> lst1 is lst2
True
```

Orders of Growth

Orders of Growth

Recall that the order of growth of a function expresses how long it takes for the function to run, and is defined in terms of the function's input sizes.

For example, let's say that we have the function `get_x` which is defined as follows:

```
def get_x(x):
    return x
```

`get_x` has one expression in it. That one expression takes the same amount of time to run, no matter what x is, or more importantly, how large x gets. This is called constant time.

The main two ways that a function in your program will get a running time different than just constant time is through either iteration or recursion. Let's start with some iteration examples!

The (simple) way you figure out the running time of a particular while loop is to simply count the cost of each operation in the body of the while loop, and then multiply that cost by the number of times that the loop runs. For example, look at the following method with a loop in it:

```
def foo(n):
    i, sum = 1, 0
    while i <= n:
        sum,i = sum + i, i + 1
    return sum
```

This loop has one statement in it `sum, i = sum + i, i + 1`. This statement is considered to run in constant time, as none of its operations rely on the size of the input. Individually, `sum = sum + 1` and `i = i + 1` are both constant time operations. However, when we're looking at order of growth, we take the maximum of those 2 values and use that as the running time. In 61A, we are not concerned with how long primitive functions, such as addition, multiplication, and variable assignment, take in order to run - we are mainly concerned with *how many more times a loop is executed* or *how many more recursive calls* occur as the input increases. In this example, we execute the loop n times, and for each iteration, we only execute constant time operations, so we get an order of growth of linear.

Here are a couple of basic functions, along with their running times. Try to understand why they have the given running time.

1. Constant

   ```
   def bar(n):
       i = 0
       while i < 10:
           n = n * 2
       return n
   ```

2. Logarithmic

   ```
   def bar(n):
       i = 1
       while n:
           i = i * 3
           n = n // 2
       return i
   ```

3. Linear

```
def bar(n):
    i, a, b = 1, 1, 0
    while i <= n:
        a, b, i = a + b, a, i + 1
     return a
```

4. Quadratic

```
def bar(n):
    sum = 0
    a, b = 0, 0
    while a < n:
        while b < n:
            sum += (a*b)
            b += 1
        b = 0
        a += 1
    return sum
```

5. Exponential

```
def bar(n):
    if n == 0: return 1
    return bar(n - 1) + bar(n - 1)
```

Required Questions

Mutability

Q1: List-Mutation

Test your understanding of list mutation with the following questions. What would Python display? Type it in the interpreter if you're stuck!

```
python3 ok -q list-mutation -u
```

Note: if nothing would be output by Python, type `Nothing`. If the code would error, type `Error`.

Relevant Topics: Mutability (https://youtu.be/VJ7rKCHgVqE)

```
>>> lst = [5, 6, 7, 8]
>>> lst.append(6)
------

>>> lst
------

>>> lst.insert(0, 9)
>>> lst
------

>>> x = lst.pop(2)
>>> lst
------

>>> lst.remove(x)
>>> lst
------

>>> a, b = lst, lst[:]
>>> a is lst
------

>>> b == lst
------

>>> b is lst
------
```

Q2: Map

Write a function that takes a function and a list as inputs and maps the function on the given list - that is, it applies the function to every element of the list.

> Be sure to mutate the original list. This function should ***not*** return anything.

```
def map(fn, lst):
    """Maps fn onto lst using mutation.
    >>> original_list = [5, -1, 2, 0]
    >>> map(lambda x: x * x, original_list)
    >>> original_list
    [25, 1, 4, 0]
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q map
```

Q3: Swap

Implement `swap`, which takes two lists and swaps their contents.

```
def swap(a, b):
    """Swap the contents of lists a and b.

    >>> a = [1, 'two', 3]
    >>> b = [4, [5, 6]]
    >>> swap(a, b)
    >>> a
    [4, [5, 6]]
    >>> b
    [1, 'two', 3]
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q swap
```

Orders of Growth

Q4: Determining Complexity

Use Ok to test your knowledge with the following questions:

```
python3 ok -q wwpd-complexity -u
```

Be sure to ask a member of course staff if you don't understand the correct answer!

Q5: Pow

Write the following function so it runs in Θ(log k) time.

> **Hint:** this can be done using a procedure called repeated squaring (https://algorithmist.com/wiki/Repeated_squaring).

```
def lgk_pow(n,k):
    """Computes n^k.

    >>> lgk_pow(2, 3)
    8
    >>> lgk_pow(4, 2)
    16
    >>> a = lgk_pow(2, 100000000) # make sure you have log time
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q lgk_pow
```

Submit

Make sure to submit this assignment by running:

```
python3 ok --submit
```

Optional Questions

Q6: Prime

Write a function that returns whether a number is prime or not in O(sqrt(n)) time, where sqrt means square root. You can assume n >= 2.

> **Hint:** you don't need to check whether every single number that is smaller than n divides n

```
from math import sqrt
def is_prime_sqrt(n):
    """Tests whether a number N is prime or not. Implement this function
    in O(sqrt(n)) time. You can assume n >= 2

    >>> is_prime_sqrt(2)
    True
    >>> is_prime_sqrt(67092481)
    False
    >>> is_prime_sqrt(524287)
    True
    >>> is_prime_sqrt(2251748274470911)
    False
    >>> is_prime_sqrt(6700417)
    True
    >>> is_prime_sqrt(44895587973889)
    False
    >>> is_prime_sqrt(2147483647)
    True
    >>> is_prime_sqrt(67280421310721)
    True
    """
    # sqrt(k) will give the square root of k as a floating point (decimal)
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q is_prime_sqrt
```