Lab 3: Midterm Review **lab03.zip (lab03.zip)**

Due by 11:59pm on Wednesday, September 13.

Starter Files

Download lab03.zip (lab03.zip). Inside the archive, you will find starter files for the questions in this lab, along with a copy of the Ok (ok) autograder.

Submit

In order to facilitate studying for the exam, solutions to this lab are released with the lab. We encourage you to try out the problems first on your own before referencing the solutions as a guide.

Note: You do not need to submit to Gradescope to receive credit for this assignment.

All Questions Are Optional

The questions in this assignment are not graded, but they are highly recommended to help you prepare for the upcoming exam. You will receive credit for this lab even if you do not complete these questions.

Suggested Questions

Walkthrough Videos

Control

Q1: Ordered Digits

Implement the function `ordered_digits`, which takes as input a positive integer and returns `True` if its digits, read left to right, are in non-decreasing order, and `False` otherwise. For example, the digits of 5, 11, 127, 1357 are ordered, but not those of 21 or 1375.

```
def ordered_digits(x):
    """Return True if the (base 10) digits of X>0 are in non-decreasing
    order, and False otherwise.

    >>> ordered_digits(5)
    True
    >>> ordered_digits(11)
    True
    >>> ordered_digits(127)
    True
    >>> ordered_digits(1357)
    True
    >>> ordered_digits(21)
    False
    >>> result = ordered_digits(1375) # Return, don't print
    >>> result
    False

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

Use Ok to test your code:

```
python3 ok -q ordered_digits
```

Q2: K Runner

An *increasing run* of an integer is a sequence of consecutive digits in which each digit is larger than the last. For example, the number 123444345 has four increasing runs: 1234, 4, 4 and 345. Each run can be indexed **from the end** of the number, starting with index 0. In the example, the 0th run is 345, the first run is 4, the second run is 4 and the third run is 1234.

Implement `get_k_run_starter`, which takes in integers `n` and `k` and returns the 0th digit of the `k` th increasing run within `n`. The 0th digit is the leftmost number in the run. You may assume that there are at least `k+1` increasing runs in `n`.

```
def get_k_run_starter(n, k):
    """Returns the 0th digit of the kth increasing run within n.
    >>> get_k_run_starter(123444345, 0) # example from description
    3
    >>> get_k_run_starter(123444345, 1)
    4
    >>> get_k_run_starter(123444345, 2)
    4
    >>> get_k_run_starter(123444345, 3)
    1
    >>> get_k_run_starter(123412341234, 1)
    1
    >>> get_k_run_starter(1234234534564567, 0)
    4
    >>> get_k_run_starter(1234234534564567, 1)
    3
    >>> get_k_run_starter(1234234534564567, 2)
    2
    """
    i = 0
    final = None
    while _______________________________:
        while _______________________________:
            _______________________________
        final = _______________________________
        i = _______________________________
        n = _______________________________
    return final
```

Use Ok to test your code:

```
python3 ok -q get_k_run_starter
```

Q3: Nearest Power of Two

Implement the function `nearest_two`, which takes as input a positive number `x` and returns the power of two (..., 1/8, 1/4, 1/2, 1, 2, 4, 8, ...) that is nearest to `x`. If `x` is exactly between two powers of two, return the larger.

You may change the starter implementation if you wish.

```
def nearest_two(x):
    """Return the power of two that is nearest to x.

    >>> nearest_two(8)    # 2 * 2 * 2 is 8
    8.0
    >>> nearest_two(11.5) # 11.5 is closer to 8 than 16
    8.0
    >>> nearest_two(14)   # 14 is closer to 16 than 8
    16.0
    >>> nearest_two(2015)
    2048.0
    >>> nearest_two(.1)
    0.125
    >>> nearest_two(0.75) # Tie between 1/2 and 1
    1.0
    >>> nearest_two(1.5)  # Tie between 1 and 2
    2.0

    """
    power_of_two = 1.0
    "*** YOUR CODE HERE ***"
    return power_of_two
```

Use Ok to test your code:

```
python3 ok -q nearest_two
```

Higher Order Functions

These are some utility function definitions you may see being used as part of the doctests for the following problems.

```
from operator import add, mul

square = lambda x: x * x

identity = lambda x: x

triple = lambda x: 3 * x

increment = lambda x: x + 1
```

Q4: Make Repeater

Implement the function `make_repeater` so that `make_repeater(func, n)(x)` returns `func(func(...func(x)...))`, where `func` is applied `n` times. That is, `make_repeater(func, n)` returns another function that can then be applied to another argument. For example, `make_repeater(square, 3)(42)` evaluates to `square(square(square(42)))`.

```
def make_repeater(func, n):
    """Returns the function that computes the nth application of func.

    >>> add_three = make_repeater(increment, 3)
    >>> add_three(5)
    8
    >>> make_repeater(triple, 5)(1) # 3 * 3 * 3 * 3 * 3 * 1
    243
    >>> make_repeater(square, 2)(5) # square(square(5))
    625
    >>> make_repeater(square, 4)(5) # square(square(square(square(5))))
    152587890625
    >>> make_repeater(square, 0)(5) # Yes, it makes sense to apply the function zero times!
    5
    """
    "*** YOUR CODE HERE ***"

def composer(func1, func2):
    """Returns a function f, such that f(x) = func1(func2(x))."""
    def f(x):
        return func1(func2(x))
    return f
```

Use Ok to test your code:

```
python3 ok -q make_repeater
```

Q5: Apply Twice

Using `make_repeater` define a function `apply_twice` that takes a function of one argument as an argument and returns a function that applies the original function twice. For example, if `inc` is a function that returns `1` more than its argument, then `apply_twice(inc)` should be a function that returns two more:

```
def apply_twice(func):
    """Returns a function that applies func twice.

    func -- a function that takes one argument

    >>> apply_twice(square)(2)
    16
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q apply_twice
```

Q6: It's Always a Good Prime

Implement `div_by_primes_under`, which takes in an integer `n` and returns an n-divisibility checker. An *n-divisibility-checker* is a function that takes in an integer k and returns whether `k` is divisible by any integers between 2 and `n`, inclusive. Equivalently, it returns whether `k` is divisible by any primes less than or equal to `n`.

Review the Disc 01 `is_prime` problem for a reminder about prime numbers.

You can also choose to do the no lambda version, which is the same problem, just with defining functions with def instead of lambda.

Hint: If struggling, here is a partially filled out line for after the `if` statement:

```
checker = (lambda f, i: lambda x: __________)(checker, i)
```

```
def div_by_primes_under(n):
    """
    >>> div_by_primes_under(10)(11)
    False
    >>> div_by_primes_under(10)(121)
    False
    >>> div_by_primes_under(10)(12)
    True
    >>> div_by_primes_under(5)(1)
    False
    """
    checker = lambda x: False
    i = ______________________________
    while ______________________________:
        if not checker(i):
            checker = ______________________________
        i = ______________________________
    return ______________________________

def div_by_primes_under_no_lambda(n):
    """
    >>> div_by_primes_under_no_lambda(10)(11)
    False
    >>> div_by_primes_under_no_lambda(10)(121)
    False
    >>> div_by_primes_under_no_lambda(10)(12)
    True
    >>> div_by_primes_under_no_lambda(5)(1)
    False
    """
    def checker(x):
        return False
    i = ______________________________
    while ______________________________:
        if not checker(i):
            def outer(______________________________):
                def inner(______________________________):
                    return ______________________________
                return ______________________________
            checker = ______________________________
        i = ______________________________
    return ______________________________
```

Use Ok to test your code:

```
python3 ok -q div_by_primes_under
python3 ok -q div_by_primes_under_no_lambda
```

Environment Diagrams

Q7: Doge

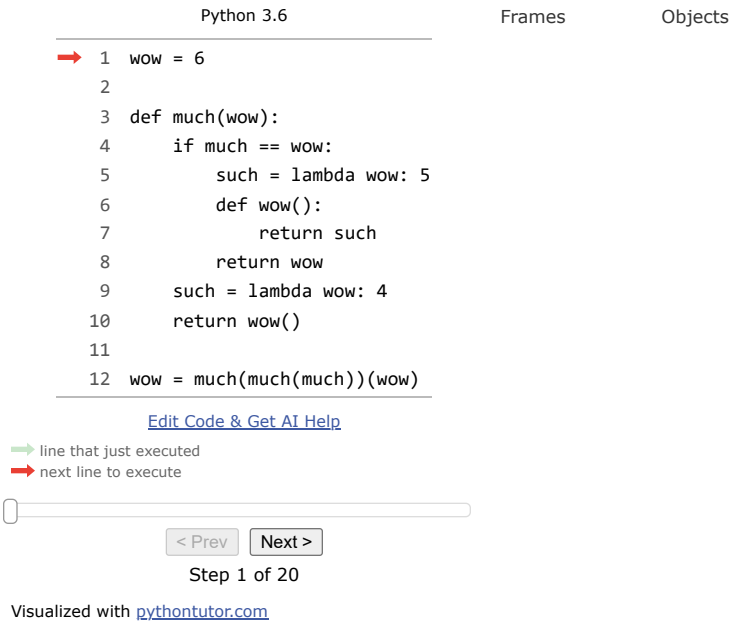
Draw the environment diagram for the following code.

```
wow = 6

def much(wow):
    if much == wow:
        such = lambda wow: 5
        def wow():
            return such
        return wow
    such = lambda wow: 4
    return wow()

wow = much(much(much))(wow)
```

You can check out what happens when you run the code block using Python Tutor (https://goo.gl/rLDpDe). Please ignore the “ambiguous parent frame” message on step 18. The parent is in fact f1.



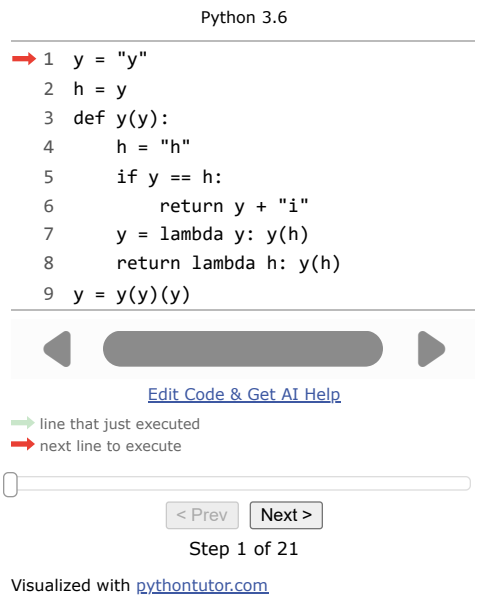
Q8: YY Diagram

Draw the environment diagram that results from executing the code below.

> Tip: Using the `+` operator with two strings results in the second string being appended to the first. For example `"C" + "S"` concatenates the two strings into one string `"CS"`.

```
y = "y"
h = y
def y(y):
    h = "h"
    if y == h:
        return y + "i"
    y = lambda y: y(h)
    return lambda h: y(h)
y = y(y)(y)
```

You can check out what happens when you run the code block using Python Tutor (https://pythontutor.com/visualize.html#code=y%20%3D%20%22y%22%0Ah%20%3D%20y%0Adef%20y%28y%29%3A%20%20%20%20%20%20%0A%2
frontend.js&py=3&rawInputLstJSON=%5B%5D&textReferences=false).



Q9: Environment Diagrams - Challenge

> These questions were originally developed by Albert Wu (http://albertwu.org/cs61a/) and are included here for extra practice. We recommend checking your work in PythonTutor (https://tutor.cs61a.org) after filling in the diagrams for the code below.

Challenge 1

Draw the environment diagram that results from executing the code below.

> Guiding Notes: Pay special attention to the names of the frames!
>
> Multiple assignments in a single line: We will first evaluate the expressions on the right of the assignment, and then assign those values to the expressions on the left of the assignment. For example, if we had `x, y = a, b`, the process of evaluating this would be to first evaluate `a` and `b`, and then assign the value of `a` to `x`, and the value of `b` to `y`.

```
def funny(joke):
    hoax = joke + 1
    return funny(hoax)

def sad(joke):
    hoax = joke - 1
    return hoax + hoax

funny, sad = sad, funny
result = funny(sad(1))
```

Challenge 2

Draw the environment diagram that results from executing the code below.

```
def double(x):
    return double(x + x)

first = double

def double(y):
    return y + y

result = first(10)
```