Lab 3 Solutions **lab03.zip (lab03.zip)**

Solution Files

Submission

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 run `python ok --submit` **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

Control

Q1: Unique Digits

Write a function that returns the number of unique digits in a positive integer.

> **Hints:** You can use `//` and `%` to separate a positive integer into its one's digit and the rest of its digits.
>
> You may find it helpful to first define a function `has_digit(n, k)`, which determines whether a number `n` has digit `k`.

```
def unique_digits(n):
    """Return the number of unique digits in positive integer n.

    >>> unique_digits(8675309) # All are unique
    7
    >>> unique_digits(1313131) # 1 and 3
    2
    >>> unique_digits(13173131) # 1, 3, and 7
    3
    >>> unique_digits(10000) # 0 and 1
    2
    >>> unique_digits(101) # 0 and 1
    2
    >>> unique_digits(10) # 0 and 1
    2
    """
    unique = 0
    while n > 0:
        last = n % 10
        n = n // 10
        if not has_digit(n, last):
            unique += 1
    return unique

# Alternate solution
def unique_digits_alt(n):
    unique = 0
    i = 0
    while i < 10:
        if has_digit(n, i):
            unique += 1
        i += 1
    return unique

def has_digit(n, k):
    """Returns whether K is a digit in N.
    >>> has_digit(10, 1)
    True
    >>> has_digit(12, 7)
    False
    """
    while n > 0:
        last = n % 10
        n = n // 10
        if last == k:
            return True
    return False
```

Use Ok to test your code:

```
python3 ok -q unique_digits
```

We have provided two solutions:

- In one solution, we look at the current digit, and check if the rest of the number contains that digit or not. We only say it's unique if the digit doesn't exist in the rest. We do this for every digit.
- In the other, we loop through the numbers 0-9 and just call `has_digit` on each one. If it returns true then we know the entire number contains that digit and we can one to our unique count.

Q2: 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


    >>> cases = [(1, True), (9, True), (10, False), (11, True), (32, False),
    ...          (23, True), (99, True), (111, True), (122, True), (223, True),
    ...          (232, False), (999, True),
    ...          (13334566666889, True), (987654321, False)]
    >>> [ordered_digits(s) == t for s, t in cases].count(False)
    0
    """
    last = x % 10
    val = x // 10
    while x > 0 and last >= x % 10:
        last = x % 10
        x = x // 10
    return x == 0
```

Use Ok to test your code:

```
python3 ok -q ordered_digits
```

We split off each digit in turn from the right, comparing it to the previous digit we split off, which was the one immediately to its right. We stop when we run out of digits or we find an out-of-order digit.

Q3: 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.

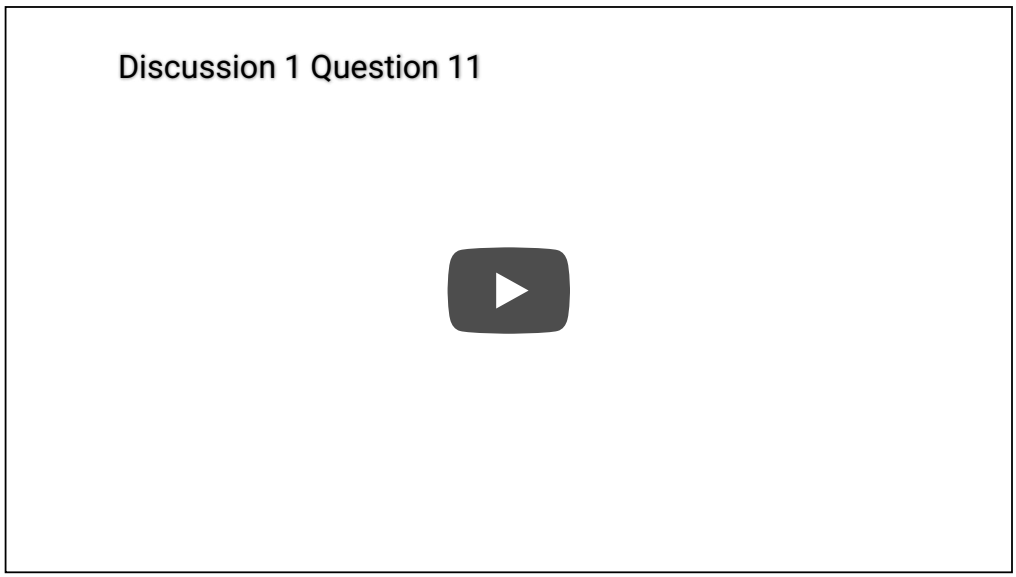
Implement `get_k_run_starter`, which takes in integers `n` and `k` and returns the 0th digit of the `k` th increasing run within `n`. The 0th digit is the leftmost number in the run. You may assume that there are at least `k+1` increasing runs in `n`.

```
def get_k_run_starter(n, k):
    """
    >>> 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 i <= k:
        while n > 10 and (n % 10 > (n // 10) % 10):
            n = n // 10
        final = n % 10
        i = i + 1
        n = n // 10
    return final
```

Use Ok to test your code:

```
python3 ok -q get_k_run_starter
```

Video walkthrough:



YouTube link (https://youtu.be/OYAKtPzq1KQ)

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):
    """Return 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
    """
    g = identity
    while n > 0:
        g = composer(func, g)
        n = n - 1
    return g

# Alternative solutions
def make_repeater2(func, n):
    def inner_func(x):
        k = 0
        while k < n:
            x, k = func(x), k + 1
        return x
    return inner_func

def composer(func1, func2):
    """Return 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
```

Solution using `composer`:

We create a new function in every iteration of the `while` statement by calling `composer`.

Solution not using `composer`:

We create a single inner function that contains the `while` logic needed to do calculations directly, as opposed to creating another function for every `while` loop iteration.

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 `double(inc)` should be a function that returns two more:

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

    func -- a function that takes one argument

    >>> apply_twice(square)(2)
    16
    """
    return make_repeater(func, 2)
```

Use Ok to test your code:

```
python3 ok -q apply_twice
```

Using `composer` from class, the body of `apply_twice` can also be written as `return composer(func, func)`.

Environment Diagrams

Q6: Doge

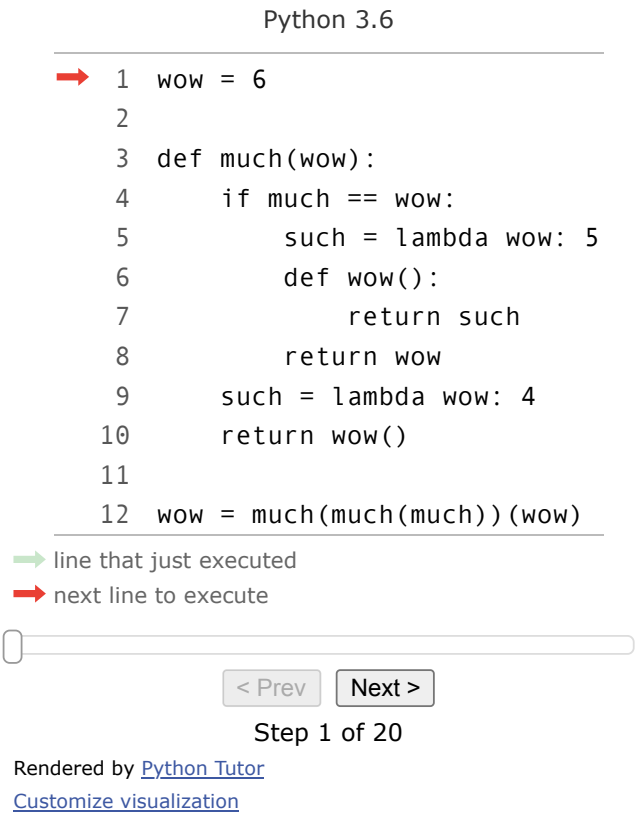
Draw the environment diagram for the following code.

```
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).



Q7: 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))
```

In the line `funny, sad = sad, funny`, we will end up with the result that the *variable* `funny` now references the function whose *intrinsic name* (the name with which it was defined) is `sad`, and the *variable* `sad` now references the function whose *intrinsic name* is `funny`.

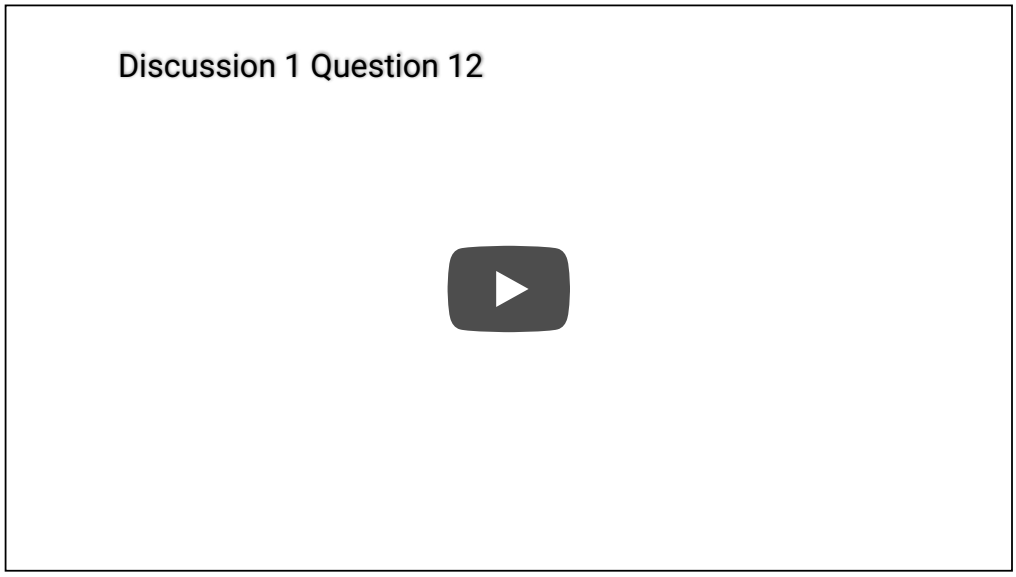
Thus, when we call `funny(sad(1))` in the last line, we will evaluate the *variable* `funny` to be the *function* called `sad` (defined from `def sad(joke): ...`). The argument we want to pass into the *function call* is `sad(1)`, where the *variable* `sad` references the *function* `funny` (defined from `def funny(joke): ...`).

Thus, our first function frame will be to evaluate `funny(1)` where this `funny` represents the intrinsic name of the actual function we're calling.

Inside of `funny`, we have the call `funny(hoax)`, which will prompt us to look up what `funny` as a variable points to. No local variable in our function frame is called `funny`, so we go to the global frame, where the *variable* `funny` points to the *function* `sad`. As a result we will call `sad` on the value `hoax`.

The return value of this function call then gets passed into a function call to `sad` (where this `sad` represents the intrinsic name of the function), and finally the return value of this function call to `sad` will be assigned to `result` in the global frame.

Video walkthrough:



YouTube link (https://youtu.be/KeQbYOyrho4)

Challenge 2

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

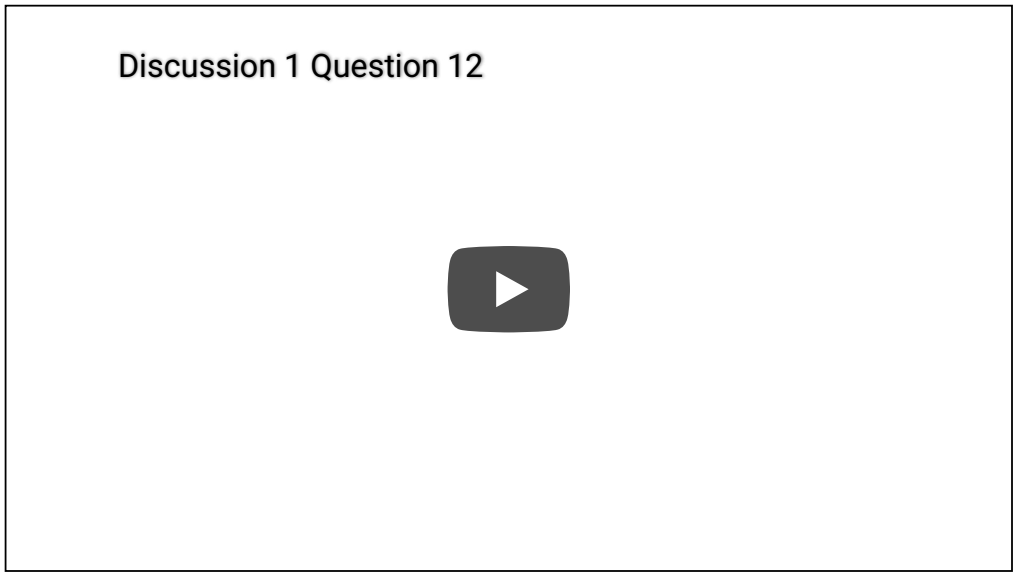
```
def double(x):
    return double(x + x)

first = double

def double(y):
    return y + y

result = first(10)
```

Video walkthrough:



YouTube link (https://youtu.be/KeQbYOyrho4)

Self Reference

Q8: Protected Secret

Write a function `protected_secret` which takes in a `password`, `secret`, and `num_attempts`.

`protected_secret` should return another function which takes in a password and prints `secret` if the password entered matches the `password` given as an argument to `protected_secret`. Otherwise, the returned function should print "INCORRECT PASSWORD". After `num_attempts` incorrect passwords are used, the secret is locked forever and the function should print "SECRET LOCKED".

For example:

```
>>> my_secret = protected_secret("oski2021", "The view from the top of the Campanile.", 1)
>>> my_secret = my_secret("oski2021")
The view from the top of the Campanile.
>>> my_secret = my_secret("goBears!")
INCORRECT PASSWORD # 0 Attempts left
>>> my_secret = my_secret("zoomUniversity")
SECRET LOCKED
```

See the doctests for a detailed example.

```
def protected_secret(password, secret, num_attempts):
    """
    Returns a function which takes in a password and prints the SECRET if the password entered matche
    the PASSWORD given to protected_secret. Otherwise it prints "INCORRECT PASSWORD". After NUM_ATTEM
    incorrect passwords are entered, the secret is locked and the function should print "SECRET LOCKE

    >>> my_secret = protected_secret("correcthorsebatterystaple", "I love UCB", 2)
    >>> my_secret = my_secret("hax0r_1") # 2 attempts left
    INCORRECT PASSWORD
    >>> my_secret = my_secret("correcthorsebatterystaple")
    I love UCB
    >>> my_secret = my_secret("hax0r_2") # 1 attempt left
    INCORRECT PASSWORD
    >>> my_secret = my_secret("hax0r_3") # No attempts left
    SECRET LOCKED
    >>> my_secret = my_secret("correcthorsebatterystaple")
    SECRET LOCKED
    """
    def get_secret(password_attempt):
        if num_attempts <= 0:
            print("SECRET LOCKED")
            return protected_secret(password, secret, num_attempts - 1)
        elif password_attempt == password:
            print(secret)
            return protected_secret(password, secret, num_attempts)
        else:
            print("INCORRECT PASSWORD")
            return protected_secret(password, secret, num_attempts - 1)
    return get_secret
```

Use Ok to test your code:

```
python3 ok -q protected_secret
```