CS 61A Fall 2017 — Midterm 1 Review

September 8, 2017

Instructions

Form a small group. Start on the first problem. Check off with a helper or discuss your *solution process* with another group once everyone understands *how to solve* the first problem and then repeat for the second problem . . .

You may not move to the next problem until you check off or discuss with another group and *everyone understands why the solution is what it is.* You may use any course resources at your disposal: the purpose of this review session is to have everyone learning together as a group.

0.1
```
>>> print(print('Welcome to'), print('CS 61A'))
```

```
Welcome to
CS 61A
None None
```

1 Functions

1.1 What would Python Display?

(a)
```
1 / 0
```

```
ZeroDivisionError
```

(b)
```
>>> def boom():
...     return 1 / 0
...
```

No error since we don't evaluate the body of the function when we define it.

(c)
```
>>> boom
```

```
<function boom at ...>
```

(d)
```
>>> boom()
```

```
ZeroDivisionError
```

1.2 What would Python display?

(a)
```
3 + 4
```

```
7
```

(b) `'3' + 4`

```
TypeError
```

(c) `'3 + 4'`

```
'3 + 4'
```

(d) `'3' + '4'`

```
'34'
```

2 Higher-Order Functions

2.1 What would Python display?

(a) `(lambda x: x(x))(lambda y: 4)`

```
4
```

(b) `(lambda x, y: y(x))(mul, lambda a: a(3, 5))`

```
15
```

2.2 Write a higher-order function that passes the following doctests.

Challenge: Write the function body in one line.

```
def mystery(f, x):
    """
    >>> from operator import add, mul
    >>> a = mystery(add, 3)
    >>> a(4) # add(3, 4)
    7
    >>> a(12)
    15
    >>> b = mystery(mul, 5)
    >>> b(7) # mul(5, 7)
    35
    >>> b(1)
    5
    >>> c = mystery(lambda x, y: x * x + y, 4)
    >>> c(5)
    21
    >>> c(7)
    23
    """


    def helper(y):
        return f(x, y)
    return helper
```

Challenge solution:

```
    return lambda y : f(x, y)
```

2.3 What would Python display?

```
>>> foo = mystery(lambda a, b: a(b), lambda c: 5 + square(c))
>>> foo(-2)


9
```

2.4 Implement `make_alternator`.

```
def make_alternator(f, g):
    """
    >>> a = make_alternator(lambda x: x * x, lambda x: x + 4)
    >>> a(5)
    1
    6
    9
    8
    25
    """


    def alternator(n):
        i = 1
        while i <= n:
            if i % 2 == 1:
                print(f(i))
            else:
                print(g(i))
            i += 1
    return alternator
```

2.5 Fill in the blanks (*without using any numbers in the first blank*) such that the entire expression evaluates to `9`.

```
(lambda x: lambda y: lambda: y(x))(3)(lambda z: z*z)()
```

2.6 Draw the environment diagram that results from running the code.

```
def dream1(f):
    kick = lambda x: mind()
    def dream2(secret):
        mind = f(secret)
        kick(2)
    return dream2

inception = lambda secret: lambda: secret
real = dream1(inception)(42)
```

https://goo.gl/TbZ1ql

3 Recursion

3.1 When you write a Recursive function, you seem to call it before it has been fully defined. Why doesn't this break the Python interpreter? Explain in haiku if possible.

Python does not care

about a function's body

until it is called

3.2 Here is a Python function that computes the `n`th Fibonnacci number. What's the domain and range of this function? Identify the three parts of this recursive program.

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)
```

The domain is in the integers and the range is in the integers. There are two base cases for checking if `n == 0` or if `n == 1`. There is one recursive case that makes two recursive calls, reducing the problem down to `fib(n - 1)` and `fib(n - 2)`, respectively.

3.3 Implement `replace`, which takes in a number `n`, a digit `old` and a digit `new`, and returns a number identical to `n`, but where every occurrence of the digit `old` is replaced with the digit `new`.

```
def replace(n, old, new):

    if n == 0:

        return 0

    last = n % 10

    rest = n // 10

    if last == old:

        return replace(rest, old, new) * 10 + new

    else:

        return replace(rest, old, new) * 10 + last
```

Tree Recursion

3.4 Implement `stairs(n)`, which takes in a number `n` and returns the number of ways to take `n` steps given that at each step you can choose to take 1, 2, or 3 steps.

```
def stairs(n):
    """
    >>> stairs(5)
    13
    >>> stairs(10)
    274
    """


    if n < 0:
        return 0
    elif n == 0:
        return 1
    else:
        return stairs(n - 1) + stairs(n - 2) + stairs(n - 3)
```

3.5 Implement `stairs(n, k)`, which takes in a number `n` and a number `k` and returns the number of ways to take `n` steps given that at each step you can choose to take any of $1, 2, \ldots, k-2, k-1$, or k steps.

```
def stairs(n, k):
    """
    >>> stairs(5, 2)
    8
    >>> stairs(5, 5)
    16
    >>> stairs(10, 5)
    464
    """


    if n < 0:
        return 0
    elif n == 0:
        return 1
    i = 1
    ways = 0
    while i <= k:
        ways += stairs(n - i, k)
        i += 1
    return ways
```

4 Exam Preparation *Extra Practice*

4.1 For each of the expressions in the table below, write the output displayed by the interactive Python interpreter when the expression is evaluated. The output may have multiple lines. If an error occurs, write "Error", but include all output displayed before the error. If a function value is displayed, write "Function".

Assume that you have started `python3` and executed the following statements:

```python
def pup(bark):
    woof = 10
    def yip(yap):
        if bark % yap == 0:
            return woof * 3
        return yap + woof
    return yip

def spot(dog):
    per = 39
    if dog > 5:
        print("pup")
    if dog > 10:
        return pup(per)

def cloud(grr):
    print(grr * 3)

woof = 9
py = woof // 3
```

```
>>> pet = spot(13)

pup

>>> print(cloud(woof + 6))

45
None

>>> pet(py)

30

>>> pet(woof)

19

>>> pup(py)

Function

>>> pet(3)

30
```

4.2 Add parentheses and single-digit integers in the blanks below so that the expression on the second line evaluates to 2017. **You may only add parentheses and single-digit integers.** You may leave some blanks empty.

```python
lamb = lambda lamb: lambda: lamb + lamb


lamb(1000)() + (lambda b, c: b() * b() + c)(lamb(2), 1)
```

4.3 Implement the `memory` function, which takes a number `x` and a single-argument function `f`. It returns a function with a peculiar behavior that you must discover from the doctests. **You may only use names and call expressions in your solution. You may not write numbers or use features of Python not yet covered in the course.**

```
square = lambda x: x * x
double = lambda x: 2 * x

def memory(x, f):
    """Return a higher-order function that prints its memories.

    >>> f = memory(3, lambda x: x)
    >>> f = f(square)
    3
    >>> f = f(double)
    9
    >>> f = f(print)
    6
    >>> f = f(square)
    3
    None
    """
    def g(h):

        print(f(x))


        return memory(x, h)

    return g
```

4.4 Implement a `counter` that returns a function which accepts digits in a given base and returns the value in base 10 after encountering `'done'`. Numbers that are not digits in the given base are ignored.

Hint: What should `parse` return?

$$\left[\left(\left[\left([(\mathbf{1})\cdot 2] + \mathbf{0}\right)\cdot 2\right] + \mathbf{1}\right)\cdot 2\right] + \mathbf{1}$$

Instructor's Hint: While this problem might seem like something you've never learned before, remember to rely on your intuition and experience with the *problem-solving process*. Revisit the doctests when you feel stuck, run through the code in your head, and ask yourself questions to make progress.

```
def counter(base):
    """Return a function which accepts digits in a given base and returns the value in base
    10 after encountering 'done'. Numbers that are not digits in the given base are ignored.

    >>> binary = counter(2)
    >>> binary('done')
    0
    >>> binary(1)(0)(1)(1)('done')         # see example above
    11
    >>> binary(1)(2)(3)(0)(1)('done')      # 2 and 3 are not digits in base 2
    5
    >>> quaternary = counter(4)
    >>> quaternary(1)(2)(3)(0)(1)('done')  # 1*(4**4) + 2*(4**3) + 3*(4**2) + 0*(4**1) + 1*1
    433
    """
    def parse(digit, total):

        if digit == 'done':

            return total

        elif digit >= base:

            return lambda x: parse(x, total)

        return lambda x: parse(x, total * base + digit)

    return lambda x: parse(x, 0)
```