Midterm 1 Walkthrough

What Would Python Dip?

Question

Assume the following code has been executed.

```
def dipping(dots):
    if print("you dip"):
        return print("i dip")
    else:
        return print(dots) or dots or print("we dip")
```

What would the Python interpreter display? If the interpreter would include a new line, please enter a new line in your answer.

Walkthrough

These questions usually aim to test our ability to understand the program execution and knowledge about the order of evaluating different expressions. It is a skill that is developed with practice and comes super handy when debugging your code. WWPD questions explore what happens when you launch the python code in interactive mode (e.g. by running `python -i dipping.py`) and start executing the lines from different parts of the question. What gets displayed back by the interpreter should go as your answers.

Glancing through the skeleton, I see that the whole body of the function is a single if-else statement, where `print("you dip")` is used as the condition. We know that `print` is a *non-pure* function, which always returns (its call expression evaluates to) `None`. However, its argument will always be printed (as long as it is valid). I can assume that `"you dip"` will always be printed when `dipping` is called, since `print(”you dip”)` is the first thing Python evaluates in the code. So as long as I keep that in mind, I can view the original code as:

```
def dipping(dots):
    if None: # print displays "you dip" and evaluates to None
        return print("i dip")
    else:
        return print(dots) or dots or print("we dip")
```

The value `None` is considered a "falsey" value in Python, which means that Python won't execute the code inside the `if` and will instead execute the code in the `else`.

To reiterate our findings (which will be useful further):

1) `print` prints the text and evaluates to `None`

2) `None` is considered falsey in Boolean expressions

This simple analysis (even without considering the value of `dots` argument) already made the future code execution much simpler. Let's start with part (a):

`>>> dipping(0)`

We now know the program always starts off by printing "you dip". Now we have to understand what is going on here:

```
return print(dots) or dots or print("we dip")
```

We can break down the execution of this line as 1) evaluate `print(dots) or dots or print("we dip")` 2) return the result of that evaluation.

`print(dots)` evaluates to `None` (falsey value) and as a *side-effect* prints `0`. So we've got `None or dots or print("we dip")`. There is `or` right after, so Python continues the evaluation further (Python cannot short-circuit because `None` is considered falsy).
`dots` is 0 (also a Falsy value) and `print("we dip")` will also evaluate to `None`, while printing `"we dip"`. So our expression looks like `None or 0 or None`, which evaluates to a single `None` (when all values are falsy, the `or` operator evaluates to the last value in the chain). Thus, the return value is `None`. The interpreter doesn't display `None` return values, so no return value will be displayed at all.

Collecting everything printed so far, we have our answer:

```
you dip # from evaluating if-statement's condition
0 # from evaluating print(dots)
we dip # from evaluating the last print call
```

Part (b) undergoes a similar analysis. We keep in mind that the interpreter has printed `"you dip"` already and similar to part (a), focus on this line:

```
return print(dots) or dots or print("we dip")
```

`print(dots)` evaluates to `None` and prints `555`. In contrast to the previous part, `dots` is a positive value now, which is truthy. So the expression looks like `None or 555 or ...` — Python actually doesn't look further since it short-circuits right here. It means that it never even gets a chance to evaluate `print("we dip")` and display its text. Reminding ourselves that `or` takes the last value, the return line looks like `return 555`, which actually gets displayed. The answer then is:

```
you dip # from evaluating if-statement's condition
555 # from evaluating print(dots)
555 # the return value, result of evaluating dipping(555)
```

Remember that `dipping(0)` got evaluated to `None`, which did not get displayed? Thus, the answer for part (c) is `None`.

Part (d) is also only concerned about the return value, not what gets displayed. Therefore, we only want to know the value of `print(-666) or -666` — the interpreter also short-circuits since negative values are truthy. Ignoring the printed values, the expression goes from `None or -666` to `-666`, which is the answer.

Part (e) asks us to write the same functionality of `dipping` function using a single-line. The original code is a simple if-statement, which only has a single `return` in both cases. Looks like a perfect candidate to be rewritten using a conditional expression:

```
if condition:
    return first_value
else:
    return second_value
# is equivalent to
return first_value if condition else second_value
```

We can do the same with the `dipping` function:

```
def dipping(dots):
    return print("i dip") if print("you dip") else (print(dots) or dots or print("we dip"))
```

Ring My Bell Tower

Question

The following code was used to generate the environment diagram below:

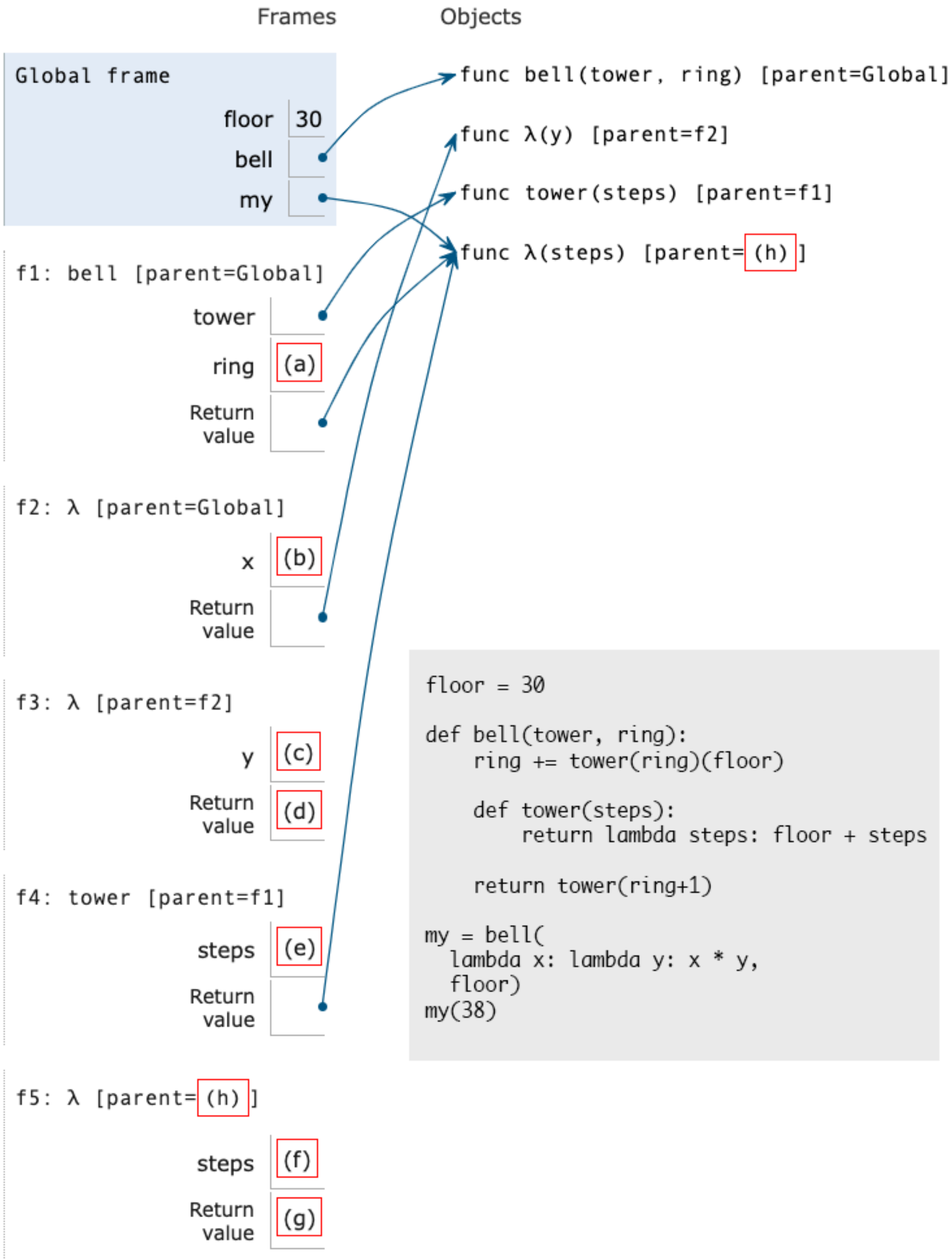
```
floor = 30
def bell(tower, ring):
    ring += tower(ring)(floor)

    def tower(steps):
        return lambda steps: floor + steps

    return tower(ring+1)

my = bell(lambda x: lambda y: x * y, floor)
my(38)
```

The environment diagram below represents the final state of the environment. The code is also provided to the right of the diagram, for convenience. Line numbers have been omitted intentionally.




```
floor = 30

def bell(tower, ring):
    ring += tower(ring)(floor)

    def tower(steps):
        return lambda steps: floor + steps

    return tower(ring+1)

my = bell(
  lambda x: lambda y: x * y,
  floor)
my(38)
```

Walkthrough

Doing well on environment diagram questions usually require a ton of practice and paying rigorous attention to all of the details. When you are given the environment that represents the final state, it is helpful to look at it for some hints, but it is better to just draw your own diagram and follow the code execution line-by-line. At the end, if your diagram arrived to the same state as the one in the question, you are likely to be correct. If not, mostly it is much easier to quickly start from scratch, instead of attempting to "fix" your current diagram.

This link (https://pythontutor.com/composingprograms.html#code=floor%20%3D%2030%0Adef%20bell%28tower,%20ring%29%3A%0A%20%20%20%20ri will take you to the Python Tutor with the code. You can launch it and follow the comments below that describe what happens on each line. Each bullet point represents one step in the environment diagram (there should be 21 total). Follow the red arrow in Python Tutor and the respective comment should tell what that line will do after it is executed.

- **Step 1, Line 1:** define a variable `floor` with value `30`.
- **Step 2, Line 2:** define a variable `bell` that refers to a function `bell` defined in the global frame. Skip the function body since the function is not being called.
- **Step 3, Line 8:** make a call to `bell` with arguments `lambda x: lambda y: x * y` and `floor`. We notice that the first argument is a lambda function that returns another lambda function. In order to not overcomplicate things, for now I can look at it as `lambda x: <something>`, because we will only care about that `<something>` when this lambda function gets called.
- **Step 4, Line 2:** create a frame for `bell` and assign `tower` to be a lambda function. Since lambda functions are expressions, it was evaluated in the global frame, before Python opened a frame for `bell`. Therefore, its parent is a Global frame. `ring` gets a value of `floor`, which was evaluated to `30`.
- **Step 5, Line 3:** we are going add something to `ring` (it better be a number, since `ring` is equal to `30` for now). The expression to be added is `tower(ring)(floor)`. According to the call expression rules, Python firstly evaluates what `tower(ring)` is, before it can call its result on the value of `floor`. The `tower` is a lambda function, so the frame that opens is going to be for lambda.

- **Steps 6-8, Line 8:** Python goes to line 8 because this is where the body of the lambda function is. It takes one argument `x`, which got the value of `ring` from Frame 1 — `30`. Its return value is a function that takes argument `y` and also does "something" (we will only care about it when we execute it). We only have to know that this lambda function was defined in the body of outer lambda, so it has `f2` as a parent frame.
- **Steps 9-11, Line 8:** the evaluation of `tower(ring)` is completed — we have figured it is a lambda function defined in `f2`. You can imagine that Python quickly goes back to line 3 and immediately calls the lambda with argument `floor`, which has the value of `30`. The frame `f3` gets created and `y` gets mapped to `30`. The return value is the result `x*y`, where `x` and `y` are both `30`, resulting in `900`. The variable `x` is not in `f3`, so it was looked up from its parent — `f2`.
- **Step 12, Line 4:** Python comes back from lambdas and updates the `ring` to be `930` (it was `30` and the result of the expression was added, `900`). Now the code defines `tower(steps)` with parent `f1` (because its body is in `f1`). Observe that the argument variable `tower` gets reassigned to this function, dropping a reference to a lambda function it had before.
- **Step 13, Line 6:** before returning, Python has to evaluate `tower(ring + 1)`. `ring + 1` would be `931` and `tower` is a function defined in the previous step.
- **Step 14, Line 4:** open a frame `f4` for `tower`, where `steps` takes up the value of `931`.
- **Step 15, Line 5:** similarly to step 13, Python firstly evaluates the result of `lambda steps: floor + steps` before it returns.
- **Step 16, Line 5:** evaluate the previous expression as a lambda function that takes argument `steps` and does something. It was defined in the body of `tower(steps)`, so it takes `f4` as its parent. This is our return value.
- **Step 17, Line 6:** come back to the line that called `tower` and bring the evaluated result. It is also going to be a return value.
- **Step 18, Line 9:** completed the execution of line 17 and assigned `my` to be a lambda function defined on line 5, with parent `f4`. Now we would like to call it with argument `38`.
- **Steps 19-21, Line 5:** open a frame for this lambda function and `steps` gets assigned to `38`. The return value of this function is `floor + steps`. Since `floor` is not in this newborn frame `f5`, Python looks it up from its parent `f4`, which does another lookup to its own parent `Global` to find the value `30`. The return value is `30 + 38`, resulting in `68`.

The diagram we got so far should be consistent with the reference diagram and we can fill in the blanks (a) - (h).

Doctor Octalpus, Reborn

Question

The standard number representation system is the decimal system, where each digit in a number represents a power of ten. The right-most digit is the ones' place, the next digit is the tens' place, etc. In the octal system, each digit in a number represents a power of eight. The right-most digit is still the 1's place, but the next digit is the 8's place, the next digit is the 64's place, etc. Each digit ranges from 0-7, so octal numbers will never contain the digits 8 or 9. To convert a number represented in octal to a number represented in decimal, each digit must be multipled by the appropriate power of eight. For example, `123` is actually `(1 * 64) + (2 * 8) + (3 * 1)`, resulting in a decimal representation of 83. The diagram visualizes the equivalence between the octal and decimal numbers:

Implement `convert_to_decimal`, which takes an octal number and returns the decimal equivalent. The octal number will always start with a non-0 digit, and the number will always be positive.

```
def convert_to_decimal(octal):
    """
    >>> convert_to_decimal(3) # (8^0 * 3)
    3
    >>> convert_to_decimal(23) # (8^1 * 2) + (8^0 * 3)
    19
    >>> convert_to_decimal(123) # (8^2 * 1) + (8^1 * 2) + (8^0 * 3)
    83
    """
    decimal = 0
    curr_place = _________ # (a)
    _________: # (b)
        curr_digit = _________ # (c)
        decimal = _________ # (d)
        curr_place = _________ # (e)
        octal = _________ # (f)
    return decimal
```

Walkthrough

If you didn't know that we typically use the decimal system and that there are other representation systems, now you know :D! Here is what we have learned from the prompt:

- In decimal, a number like 83 is constructed like `83 = 8 * 10^1 + 3 * 10^0 = 80 + 3`
- The same 83, but in the octal system would look like 123. Because `123 = 1 * 8^2 + 2 * 8^1 + 3 * 8^0 = 64 + 16 + 3 = 83`
- Observe that in any number representation system, the ones' place (the rightmost digit) is always multiplied by 1.

Doctests are self-explanatory here, but it always pays off to test our understanding with them. Now we can get started with the problem-solving!

- This is a problem about digit manipulation, so we use our best friends `n % 10` and `n // 10` to deal with it.

The code starts with defining `decimal`, which looks to be our final answer. Then we meet blank (a) that sets a value to `curr_place` variable. No idea what to do with it, so let's move on.

For blank (b), it is a while-loop, since a single execution of the if-statement (from what we know so far, nothing else in Python could be ended with `:`) can't process all digits from the `octal`. Since we use `octal // 10` to "move" along the digits of the number, we will stay in the while-loop until there are digits in `octal`. In other words:

```
while octal > 0:
```

To make sure we don't enter an infinite loop, we can put `octal // 10` into blank (f) to advance through the digits and eventually trim `octal` down to `0`.

Blank (c) is less mysterious than its "curr" sibling from blank (a). It asks for a current digit in octal (which we will definitely use to convert the number back to decimal), so `current_digit = octal % 10`. Observe that on every iteration, `current_digit` will be equal to the rightmost digit.

Now we have to update the decimal. According to the formula, we multiply the rightmost digit by the appropriate power of 8. Every time we advance to the next digit (which means for every iteration in the while-loop), the power of 8 increases by 1. Do we have a variable to keep track of the power of 8? Yes — `curr_place`! On the first iteration, the rightmost digit should be multiplied by `8^0 = 1`, so our initial value of `curr_place` on the blank (a) will look like:

```
curr_place = 1
```

and accordingly, the blank (d) will be:

```
decimal = decimal + (curr_digit * curr_place)
# compute the product of the rightmost digit with the appropriate power of 8
# and add that to our result so far (decimal)
```

Finally, update the `curr_place` to increase its power of 8 in the blank (e). We can achieve that with:

```
curr_place = curr_place * 8
```

Our final solution looks like:

```
def convert_to_decimal(octal):
    decimal = 0
    curr_place = 1
    while octal > 0:
        curr_digit = octal % 10
        decimal = decimal + (curr_digit * curr_place)
        curr_place = curr_place * 8
        octal = octal // 10
    return decimal
```

Here is a couple of alternative solutions:

```
def convert_to_decimal(octal):
    decimal = 0
    curr_place = 0
    while octal > 0:
        curr_digit = octal % 10
        decimal = decimal + (curr_digit * (8**curr_place))
        curr_place = curr_place + 1
        octal = octal // 10
    return decimal

def convert_to_decimal(octal):
    decimal = 0
    curr_place = 1
    while octal > 0:
        curr_digit = (octal % 10) * curr_place
        decimal = decimal + curr_digit
        curr_place = curr_place * 8
        octal = octal // 10
    return decimal
```

Forbidden Digits

Question

Implement `forbid_digit`, a higher-order function which takes two arguments, a function `f` and a digit `forbidden`, and returns another function. If the returned function is passed a number where the digit in the 1s place is equal to the forbidden digit, it should return the result of calling the given function on the number without that final digit. Otherwise, it should return the result of calling the given function on the number.

```
def forbid_digit(f, forbidden):
    """
    >>> g = forbid_digit(lambda y: 200 // (y % 10), 0)
    >>> g(11)
    200
    >>> g(10)
    200
    >>> g = forbid_digit(lambda x: f'{x}a', 6)
    >>> g(61)
    '61a'
    >>> g(66)
    '6a'
    >>> g = forbid_digit(g, 3)
    >>> g(43)
    '4a'
    >>> g(63)
    '0a'
    >>> g(44)
    '44a'
    """
    def forbid_wrapper(n):
        if ________: # (a)
            ________ # (b)
        else:
            ________ # (c)
    ________ # (d)
```

Walkthrough

Let's make a summary of the problem statement:

1) `forbid_digit` is a HOF and its return value is a function that accepts a single number. Skeleton confirms that there is another function `forbid_wrapper` in `forbid_digit` and it takes one argument — an ideal candidate for our return value.

2) With regards to that returned function, if it gets a number `n` where the rightmost digit is equal to `forbidden`, it should return the result of calling `f` on `n`, but with the rightmost digit removed)

3) Otherwise (if the rightmost digit is not `forbidden`), just call return `f(n)` (don't change/remove any digits)

It is always useful to confirm our understanding against the doctests:

```
>>> g = forbid_digit(lambda y: 200 // (y % 10), 0) # g is a function now
>>> g(11) # the rightmost digit is 1, 1 != 0, so just apply lambda on 11
200 # 200 // (11 % 10) = 200 // 1 = 200
>>> g(10) # the rightmost digit is 0, so apply lambda on 1, not 10
200 # 200 // (1 % 10) = 200 // 1 = 200
```

```
>>> g = forbid_digit(lambda x: f'{x}a', 6)
>>> g(61) # the rightmost digit is 1, 1 != 0, apply lambda on 61
'61a' # format string simply puts 61 instead of x, resulting in 61a
>>> g(66) # the rightmost digit is 6, apply lambda on just 6
'6a'
```

```
>>> g = forbid_digit(g, 3) # argument g here is a function from the previous set of doctests
>>> g(43) # the rightmost digit is 3, so remove it and return g(4)
'4a'
>>> g(63) # remove 3 just like previous doctest, and return g(6)
'0a'
>>> g(44) # simply return g(44)
'44a'
```

- This doctest might have been tricky, because we use `g` from the previous call to `forbid_digit`. We should remember that it was a function that removed the rightmost digit of the argument if it was equal to `6` and attached `a` to the result.
- Alternatively, you can look at the first line `g = forbid_digit(g, 3)` as `g = forbid_digit(forbid_digit(lambda x: f'{x}a', 6), 3)`. Now you know that `f` is something more complicated than a simple lambda function.

- Particularly, when `g(63)` is called, we firstly get rid of `3` and call `f` function on `6`. In this case, `f` is a function from the previous set of doctests, the result of `forbid_digit(lambda x: f'{x}a', 6)`. In this case, the rightmost digit of `6` is equivalent to `forbidden`, which results in calling lambda on `0`, hence the final answer is `0a`.

Now we are ready to get started. Per point (1) from our prompt summary, go ahead and put `return forbid_wrapper` into blank (d).

The body of `forbid_wrapper` is a single if-statement and our summary points (2) and (3) actually read like a if-else logic: "if it gets a number `n`...", "otherwise (if the...".

Let's use blank (a) to compare the rightmost digit of `n` with forbidden, so we can put something like:

```
n % 10 == forbidden # n % 10 is equal ot the rightmost digit
```

- Note that since `forbid_wrapper` is defined inside `forbid_digit`, it has access to `f` and `forbidden` values. For example, if we define `g = forbid_digit(lambda x: f'{x}a', 6)`, all future calls of `g` will use `6` as `forbidden` value.

If our condition is true, in blank (b) we should apply the `f` function on the `n` with its rightmost digit removed. Floor dividing `n` by 10 achieves the "removal" effect:

```
return f(n // 10)
```

If the condition happened to be False, we simply apply `f` on `n` on the blank (c) and we are done!

```
return f(n)
```

Our complete solution looks like:

```
def forbid_digit(f, forbidden):
    def forbid_wrapper(n);
        if n % 10 == forbidden:
            return f(n // 10)
        else:
            return f(n)
    return forbid_wrapper

# you could also flip the condition and the return values
def forbid_digit(f, forbidden):
    def forbid_wrapper(n);
        if n % 10 != forbidden:
            return f(n)
        else:
            return f(n // 10)
    return forbid_wrapper
```

Part (e) is similar to part (e) of the Question 1. Actually, the structure of our code is also similar to `dipping` — it is a single if-statement. So we once again can utilize the conditional expression:

```
if condition:
    return first_value
else:
    return second_value

# is equivalent to
return first_value if condition else second_value
```

Then the answer for part (e) is:

```
def forbid_digit(f, forbidden):
    return lambda n: f(n // 10) if n % 10 == forbidden else f(n)
```

- See that the return value is also a function (lambda) that accepts a single argument `n`. In general, whenever we need to define a function and have only a single line to do it — we use lambda. If-else logic can also fit-in with the help of conditional expression.

The Floor is Lava

Question

Implement `lava_hopper`, a function that "hops" from one number to the next computed number and tries to avoid any number detected as "lava". When it does land on "lava", it steps backwards by one number until it finds a non-lava number and then keeps hopping. The function takes four arguments: `start_number` (the initial number), `goal_number` (the target number), `next_hop` (a function that computes the next number based on the current), and `is_lava` (a function that returns a boolean indicating if a number is lava), and it returns the minimum number of hops required to get from `start_number` to at least `goal_number`. The number of hops does not include steps backwards. If either the `start_number` or `goal_number` spots are lava, it returns the string "No lava allowed there!".

For example, consider this call:

```
lava_hopper(1, 8, lambda x: x * 2, lambda x: x == 4)
```

The function starts from the number 1 and then hops to the numbers 2, 4, realizes that's lava, steps back to 3, hops to 6, hops to 12, and returns 4 (the number of hops required to get to/past 8). Notice that depending on the functions passed in for `next_hop` and `is_lava`, it is possible for a correct `lava_hopper` implementation to result in an infinite loop.

```
def lava_hopper(start_number, goal_number, next_hop, is_lava):
    """
    >>> # hops from 1->2, 2->4, 4->8
    >>> lava_hopper(1, 8, lambda x: x * 2, lambda x: False)
    3
    >>> # hops from 1->2, 2->4, steps to 3, hops 3->6, hops 6->12
    >>> lava_hopper(1, 8, lambda x: x * 2, lambda x: x == 4)
    4
    >>> # hops from 1->2, 2->4, 4->8, steps to 7, then 6, then 5, hops to 10
    >>> lava_hopper(1, 10, lambda x: x * 2, lambda x: 6 <= x <= 8)
    4
    >>> # hops from 3->6, 6->12, steps to 11, hops 11->22
    >>> lava_hopper(3, 20, lambda x: x * 2, lambda x: x % 10 == 2)
    3
    >>> lava_hopper(1, 8, lambda x: x * 2, lambda x: x == 1)
    'No lava allowed there!'
    >>> lava_hopper(1, 8, lambda x: x * 2, lambda x: x == 8)
    'No lava allowed there!'
    """
    if __________: # (a)
        return 'No lava allowed there!'
    num_hops = 0
    while __________: # (b)
        _________________: # (c)
            __________ # (d)
        start_number = __________ # (e)
        __________ # (f)
    return num_hops
```

Walkthrough

Even though there is quite a lot to digest in the problem description, what is actually happening while (pun intended) we hop is pretty manageable. After carefully reading over the prompt for at least two times, what we have learned is:

1. We start at `start_number` and "hop" until we reach or **exceed** the `goal_number`.
2. The return value is the number of hopes we made. We calculate the number we are going to hop to using the `next_hop` function.
3. If during the hop, we step on "lava" number (checked using `is_lava` function), then we step back by 1 until we get to a number that is not lava. From there, we continue our hops as usual. Steps back are NOT considered as hops.
4. If either start or goal is already "lava", we return "No lava allowed there!"

The doctests here are provided with explanations and we can just confirm our understanding with them.

So the problem seems "mechanical", where you just have to do the hops as described, while taking care of conditions about `goal_number` or "lava". Quick scan of the skeleton shows us that we will use a while-loop to go over the hops and use `num_hops` variable to accumulate the result.

Blank (a) asks for a condition that we use to return 'No lava allowed there!'. As we know from our prompt notes, we do that if either `start_number` or `goal_number` is "lava":

```
if is_lava(start_number) or is_lava(goal_number):
    return "No lava allowed there!"
```

- Some students tried using `is_lava == start_number or is_lava == goal_number` as the condition, but we have learned both from the statement and doctests that *`is_lava`* is a function returning True/False, therefore it has to be called on numbers and its result checked for True value. Checking function against an integer for equality would never work.

Moving on, we need a while-loop condition now. We know that we use loops to perform the same operation multiple times, so probably it will be used to handle each "hop". Do we know until when we hop? From our notes, until we hit or go beyond `goal_number`. So we can put something like:

```
while start_number < goal_number:
```

- You might be tempted to use `<=` here, but if you consider a test like `lava_hopper(1, 1, lambda x: x + 1, lambda x: False)`, where start and goal are already equal, you see that you don't need to do any hops here (the answer is 0). However, using `<=` will launch the while loop iteration anyway, and it is likely that you will increase your number of hops there (since there is no other space for that). You could also catch this issue after finishing your solution and testing it (you would see that you overcount).

Whenever I write a while loop, I really like making sure it won't become an infinite loop (can be a good habit!). Instead of going to blank (c), let's focus on line (e), since it affects the variable that we use in the while-loop's condition (`start_number`). From the prompt, we know that `next_hop` is what we use to advance forward to the `goal_number` . So `start_number = next_hop(start_number)` looks to be a reasonable candidate. Defaulting to good old `start_number += 1` would not work here, since we utilize a custom lambda function to iterate forward rather than usual incrementing we use in assignments. That is why it is crucial to carefully read over the prompt first!

For blank (c), we see that it ends with `:` and is followed up by indented blank (d). From what we have learned so far in Python, what usually ends with `:`? While loops and if statements. The former is used to repeatedly perform some operations while the condition is true, when the latter also does something if the condition is true, but only once. Maybe this is where we can handle the logic of stepping into the lava. We know that if we are on the number that is "lava", we have to take a step backwards on our counter variable (`start_number`). So for blanks (c) and (d), something like this might work out:

```
if is_lava(start_number):
    start_number -= 1
```

- Looks good! However, if we are more careful, we notice that making a step back once does not necessarily put us outside of the lava right away. We might have to take more steps backwards, until we are not on the "lava" number. Single execution of the if-statement does not do that, but we can use the while-loop:

```
while is_lava(start_number):
    start_number -= 1 # we will keep decrementing start_number until we are in the lava
```

Finally, we have an empty line, blank (f). In the heat of solving the problem, it is common to forget about actually counting the answer! Every iteration in the outer while-loop embodies a single hop (note that the second while-loop does not touch the number of hops since step backs do not count as hops per the prompt), so we can put `num_hops += 1` on the blank (f). Here is our complete solution just for reference:

```
def lava_hopper(start_number, goal_number, next_hop, is_lava):
    if is_lava(start_number) or is_lava(goal_number)>: # (a)
        return 'No lava allowed there!'
    num_hops = 0
    while start_number < goal_number: # (b)
        while is_lava(start_number): # (c)
            start_number -= 1 # (d)
        start_number = next_hop(start_number) # (e)
        num_hops += 1 # (f)
    return num_hops
```

Part (g) asks us to come up with arguments that result in an infinite-loop. With such input, we would get stuck forever iterating in the while-loop. When does that happen? If we keep satisfying the loop condition every time. So `start_number` should never reach the `goal_number` . We unfortunately can't make `goal_number` a "lava" number, since it would return "No lava allowed there!", before going to the while oop. But we can come up with a `next_hop` function that never lets us move:

```
lava_hopper(1, 2, lambda x: 1, lambda x: False) # no "lava" numbers, but we will always stay at 1
```

- Or you can simply put a "lava" barrier in front of the goal:

  ```
  lava_hopper(1, 5, lambda x: x + 1, lambda x: x == 3) # We can never go past 3 to reach 5
  ```

- Or make your hopper hop backwards:

  ```
  lava_hopper(1, 2, lambda x: x - 1, lambda x: False)
  ```

Curry Up Now

Question

The function `order_meal` takes three arguments, `item_price` , `item_quantity` , and `ordered_at` , and either returns the total cost of the meal or returns "Wait!" if the meal was not ordered between business hours. Only the doctests are shown below, as the implementation is not necessary for completing the question.

```
def order_meal(item_price, item_quantity, ordered_at):
    """
    >>> order_meal(5.99, 5, 11)
    29.95
    >>> order_meal(9.99, 5, 20)
    49.95
    >>> order_meal(8.99, 5, 7)
    'Wait!'
    """
    # Code intentionally omitted
```

Implement `curry_up_now`, a function that curries `order_meal` into a chain of three functions that each take a single argument. Once the third function is called, it should attempt to order the meal and print out the result. If the meal was successfully ordered during business hours, it should then return another curried function that can re-order the same item with a 50% discount.

```
def curry_up_now(item_price):
    """
    >>> curry_up_now(2.99)(2)(15)
    5.98
    <function <lambda>>
    >>> lunch_special = curry_up_now(8.99)
    >>> lunch_special(5)(11)
    44.95
    <function <lambda>>
    >>> lunch_special(3)(13)(2)(14)
    26.97
    8.99
    >>> no_discount = curry_up_now(10.99)(4)(7)
    Wait!
    >>> print(no_discount)
    None
    """
    def order_quantity(item_quantity):
        def by(ordered_at):
            result = ________ # (a)
            ________ # (b)
            ________: # (c)
                return ________ # (d)
        return by
    return order_quantity
```

Walkthrough

The problem statement was corrected according to the clarification given during the exam.

First of all, we have to understand the possible return values of the `order_meal` function. Here is the summary you might have come up with:

- If it returns the total cost, it seems to be just multiplying `item_price` with `item_quantity`, which conceptually makes perfect sense as the total cost of the order
- If it returns "Wait!", it is likely that `ordered_at` was the time during "business hours". We do not know what exactly these "business hours" are.
- The implementation is omitted, so we will only rely on the connection between arguments and return value. Conceptually, we can assume that only `ordered_at` should impact on whether the return value is the total cost or "Wait!".

Now, what about `curry_up_now`?

- It curries `order_meal` into a chain of three functions, which means it will wrap a call to `order_meal` into three function, nested into each other. Essentially, we aim to transform the three-argument function `order_meal` into a chain of three functions, where each one accepts a single argument. It allows for code like this: `curry_up_now(2.99)(2)(15)`.
- After the third argument (`ordered_at`) is fed in, it attempts to order the meal and print the result (calls `order_meal` and prints its return value). If the order was successful (got total cost as a result of calling `order_meal`), the function should return *another* chain of **two** functions that allows to order the same item (means the `item_price` from the original call was "saved", that is why the return value is a chain of two functions, since it only needs `item_quantity` and `ordered_at`) for 50% discount.

Doctests help a lot here to clarify what is going on:

```
>>> curry_up_now(2.99)(2)(15)
5.98 # order was between "business hours", so we get 2.99 * 2 = 5.98 printed
<function <lambda>> # returned lambda function that allows for a discounted order
# sadly we cannot call this lambda since the original curry_up_now result was
# not assigned to any variable to capture the lambda

>>> lunch_special = curry_up_now(8.99) # we "set" 8.99 to be the item price
# lunch_special is now a function that can curry-in two arguments
>>> lunch_special(5)(11) # we feed in the item_quantity of 5 and ordered_at of 11
44.95 # 8.99 * 5 = 44.95 -- the order is successful
<function <lambda>> # function that allows discounted order that was not captured again
```

- These doctests showcase how currying happens and also show that `curry_up_now` *prints* the total cost, but *returns* the lambda function.

```
>>> lunch_special(3)(13)(2)(14)
26.97
8.99
```

- This one is tricky. First of all, remember that the value of `lunch_special` is still `curry_up_now(8.99)`, where we fixed the item price to be `8.99`. So it remains to be a chain of two functions, each accepting `item_quantity` and `ordered_at`.
- `lunch_special(3)(13)` is a single order, made for item with `item_price = 8.99`, `item_quantity = 3`, `ordered_at = 13`. The order was successful since we printed `26.97` on the next line (`26.97 = 8.99 * 3`).
- We can see that the expression `lunch_special(3)(13)` is followed up by two calls `(2)(14)`, which reaffirms us that the order was successful and the expression was evaluated to be a chain of two functions, allowing for a discounted order. Indeed, `8.99 * 2 = 17.98`, but instead there is `8.99` on the next line, which is exactly 50% discount for `17.98`.

```
>>> no_discount = curry_up_now(10.99)(4)(7) # make a new call with three arguments
Wait! # the order was not during business hours, so "Wait!" is printed instead of the total cost
>>> print(no_discount)
None
# it is None because the order was unsuccessful,
# we did not get a function for making a discounted order :(
```

After a long journey with doctests, we are fortunate that the skeleton already has the currying structure of three functions, where:

- the `curry_up_now` takes argument `item_price` and returns `order_quantity` function,
- the `order_quantity` function takes `item_quantity` as argument and returns `by`
- the `by` function has `ordered_at` as argument and returns what we are going put in blank (d). But let's start from blank (a).

We know that `order_now` (for a regular price, not discounted) must be called somewhere in `curry_up_now`. Any blank from (a) - (c) can achieve that.

We also know that the result of `order_now` will be checked for whether it equals a total cost (in which case we return a lambda that for a discounted order) or a string "Wait!" (in that case, return `None`). This conditional logic might incline us to put the `order_now` call into blank (c). Something like:

```
if order_now(item_price, item_quantity, ordered_at) != "Wait!": # blank (c)
        return _________ # blank (d), curried function
# this is the end of function by, which means otherwise it returns None
```

However, the result of `order_now` must also be printed. We need to store its result somewhere to both print it and compare against `"Wait!"`. There is already a variable given to us — `result`! Let's use blank (a) for `order_now` call:

```
result = order_now(item_price, item_quantity, ordered_at)
```

As mentioned above, the result should be printed. Blank (b)?

```
print(result)
```

Our previous if-statement in blank (c) changes into:

```
if result != "Wait!":
```

The moment of truth! We need to come up with a function (also using only a single line, hi lambda!) that calls `order_meal` with discounted price and also prints its result 😱

Let's figure it out step-by-step. A call to `order_meal` for half a price should be something like:

```
order_meal(item_price * 0.5, item_quantity, ordered_at)
```

Now we need to ensure that the `item_price` used above is from the original call to `curry_up_now` (i.e. in the environment diagram, `item_price` value should come from the frame of `curry_up_now`). The remaining two arguments must be curried-in, so we need to chain two lambdas together:

```
lambda item_quantity: lambda ordered_at: order_meal(item_price * 0.5, item_quantity, ordered_at)
```

Cool! The only problem is where to print the result of `order_meal`. Well, what if...

```
lambda item_quantity: lambda ordered_at: print(order_meal(item_price * 0.5, item_quantity, ordered_at))
```

- To reiterate, this expression is a chain of two functions which calls `order_meal` with a discounted price, prints its result (either the total cost or "Wait!") and returns `None` (what a call to `print` evaluates to).

Here is the full solution for reference:

```
def curry_up_now(item_price):
    def order_quantity(item_quantity):
        def by(ordered_at):
            result = order_now(item_price, item_quantity, ordered_at)
            print(result)
            if result != "Wait!":
                return lambda item_quantity: lambda ordered_at: order_meal(item_price * 0.5, item_quantity, ordered_at)
                # otherwise return None
        return by
    return order_quantity
```

Part (e) asks us to condense this tongue-twister into a single line. It means we need to perform all of the following in the single line of code:

```
result = order_now(item_price, item_quantity, ordered_at)
print(result)
if result != "Wait!":
    return lambda item_quantity: lambda ordered_at: order_meal(item_price * 0.5, item_quantity, ordered_at)
```

First of all, we can try replacing `result` on lines 2 and 3 above with a call to `order_now`, so there is no need to have a variable `result`. Something like:

```
print(order_now(item_price, item_quantity, ordered_at))
if order_now(item_price, item_quantity, ordered_at) != "Wait!":
    return lambda item_quantity: lambda ordered_at: order_meal(item_price * 0.5, item_quantity, ordered_at)
```

Okay, one line is gone. The conditional expression always rescued us in previous problems, when we needed a one-liner, so let's try utilize it here as well:

```
print(order_now(item_price, item_quantity, ordered_at))
return lambda q: lambda h: order_meal(item_price * 0.5, q, h) if order_now(item_price, item_quantity, ordered_at) != "Wait!" else None
```

- The arguments of the lambda functino that we return were changed to `q` and `h` to not get confused with the call that happens in the condition.

Finally, we need a way for both of these lines to execute in a single line and the last expression (conditional one) should be returned. Since a call to `print` returns `None`, which is considered falsey value, we can put `or` between a call to `print` and the conditional expression:

```
return print(order_now(item_price, item_quantity, ordered_at)) or ( (lambda q: lambda h: order_meal(item_price * 0.5, q, h)) if order_
```

A call to `print` will display the result of `order_now` call and then will be evaluated to `None`. The return expression will look like:

```
return None or ( (lambda q: lambda h: order_meal(item_price * 0.5, q, h)) if order_now(item_price, item_quantity, ordered_at) != "Wait
```

Such expression will inevitably return the result of evaluating the conditional expression, which is either `None` (if our second call to `order_now` resulted in "Wait!") or the chain of two functions that make a discounted order.