Fall 2018 MT1 - Q6a

```
1 def repeat(k):
2     return detector(lambda n:False)(k)
3 def detector(f):
4     def g(i):
5         if f(i):
6             print(i)
7         return detector(lambda n:n==i or f(n))
8     return g
```

func1

func2

```
def func1(n):
    return False
```

```
def func2(n):
    return (n==i) or f(n)
```

by boolean expression evaluation we know that we will always first check if n=i and only if that is false will it call f(n)

① For the sole purpose of understanding the solution I'm going to give a name to each of the lambda functions and write out what the function would look like if it wasn't a lambda function.

② Let's examine what is happening on line 2.

`return detector(lambda n:False)(k)` → Think about what detector(func1) returns... it just returns g (draw out env. diagram to see this!) so now k is the argument to g (so it is the 'i')

↑ we are calling detector because we need to somehow enter the body of detector since that is where the meat of our solution lies (keep in mind that detector is also a higher order function because it returns g which is another function that is defined within the body of detector)

func1 → this is the argument to detector (so it is the 'f')

③ Following from the return call (line2) discussed above, we know we will end up at line 4. So, let's examine what happens in the body of g.

`def g(i):` → again following from above, we see that i is going to be some # (whatever # was passed in as an argument to the repeat function)

`if f(i):` → again from above, we see f is the lambda function (func1)

`print(i)` → we only reach this line if func1 returns true (but based off our definition of func1 above, we see that it will never return true. So, we will need to update this function to return true whenever an element has been repeated → so we know we will need to update the parameter that is passed into the detector function)

`return detector(lambda n:n==i or f(n))`

func2

↓ at this point, we have analyzed the input argument to repeat and now want to prepare for the next argument that will be passed in

we want to call detector again (since it is the function that can print or not print appropriately) however, we don't want to use func1 since we saw how that will always return false (only works for first # since first # can never be a repeat). So, func2 has to be a function that returns true if a # has been repeated before & false if a # has never been repeated before.

④ Finally, we can analyze the body of func2 to see how/why it only returns true if there is a repeated number.

(lambda n: n == i or f(n))

func2

```
def func2(n):
    return (n==i) or f(n)
```

these are equivalent expressions, but I will use the one on the right to explain for the purpose of clarity

```
def func2(n):
    return (n==i) or f(n)
```

→ n is the # that is passed in as an argument to f (argument of detector) → so, n is going to be the # passed into repeat (after the first one)

[repeat(3)(2)(3)]
↓ this will be n for func1

what is i?
i is going to be whatever # was passed into the repeat function on the previous call
(this what the video meant by the term "cache" — pretty much just means you are able to use a value that was a parameter previously. If that's more confusing, then don't worry about the word cache & instead just look at the environment diagram (on pythontutor). we are using the idea of scoping. You will see that there is no i defined in the frame of the lambda function, however, we know that we can always look to a function's parent when this happens & in accessing lambda's parent you will have access to the previous input to the function repeat. So, this covers the scenario where two #s are the same consecutively. However, it doesn't cover the case where a repeat does not appear immediately after the # (ie. repeat(7)(6)(5)(7)).

→ This covers the case of catching repeats when there are other #s in between them. To do this we need to access not just the last # passed into repeat but all the #s that had previously been passed in. After this call, f is updated to this lambda function. However, during this call, f is still the "old" f or the f that was previously defined (meaning its parent frame includes the 2nd to last value of i instead of the last value of i). So, in this way we are able to use this call to access all the previous values of i.
(again I recommend opening up pythontutor and stepping through all the steps for 3 or 4 diff inputs to see visually how this works if its still confusing
↳ a good example to try would be repeat(1)(2)(2)(1)