CS61A Lecture 8

Friday, September 13th, 2019

Currying

Currying is the technique of translating the evaluation of a function that takes multiple arguments into evaluating a sequence of functions, each with a single argument. For example:

```
>> make_adder(2)(3)
5
>>> add(2,3)
5
```

These two functions do the same thing. It might be beneficial to do make_adder, because in large programs, we may only know one of the parameters early.

We can generalize the technique of currying with either of the following formats:

```
curry2 = lambda f: lambda x: lambda y: f(x,y)

def curry2(f):
    def g(x):
        def h(y):
            return f(x,y)
        return h
    return g
```

In this case, it may be easier to just understand each of these lambda functions and what they do, rather than drawing an environment diagram, since assignments aren't changed.

Decorators

Here's another use for higher order functions. Let's say we have a function `squared_up_to`:

```
def square_up_to(x):
    total = 0
    while n > 0:
        total, n = total + square(n), n - 1
    return total
```

The second code prints 14 when called on 3, but how does it do that? Would it be nice to see exactly what it was doing? We can do this with a **decorator**.

```
def trace1(f):
    def traced(x):
        print('calling', f, ' on', x)
```

```
            return f(x)
        return traced
```

> You can also add `.__name__` at the end of `f` to see just the name instead of the function address.

It is called `trace1` because it takes one argument. To trace a function, you assign the name of the function you want to trace to the return value of calling `trace1` on the function.

```
square = trace1(square)
square_up_to(3)
```

You can also just use the following decorator instead of rebinding `square`:

```
@trace1
def square(x):
    return x*x
```

This tells Python to trace the function, and bind the name `square` to a traced version of `square`. Tracers aren't built into Python. The only thing built is the decorator, which calls higher-order functions.

With 61A projects, a tracer is already built for you in the `ucb.py` file. Use it if you're stuck!

Review

Implementing Functions

This section is a review for the latter half of most 61A papers.

> **Example question**
>
> ```
> def remove(n, digit):
>     """Return all digits of non-negative N that are not DIGIT, for some non-negat
>
>     >>> remove(231,3)
>     21
>     >>> remove(243132,2)
>     4313
>     """
>     kept, digits = 0, 0
>     while _____________:
>         n, last = n // 10, n % 10
>         if ______________:
>             kept = _______
>             digits = _____
>     return _______________
> ```

How to approach a problem like this

1. Verify your understanding of the problem. Check against the doctests to ensure you are implementing the function you think you're doing. Pick a simple example you can focus on when implementing the function.
2. Now read the template and try to infer the key ideas. This code initializes on kept and digits, and try to guess what you should do with those names.
3. When you read the template like this, you're going to find one of two things.
   - The template is helpful. Use it.
   - The template is annoying, confusing or otherwise not helpful. In this case, implement without the template, then change your implementation to match the template.
4. Ask questions why certain variables and lines exist in the function.
5. Annotate names with values from your chosen example from earlier, such as 231 and 3.
6. Now double check to ensure you wrote the code correctly. Check your solution with other examples.

Correct answer

```
def remove(n,digit):
    kept, digits = 0, 0
    while n > 0:
        n, last = n // 10, n % 10
        if last !== digit:
            kept = kept + last*10**digits
            digits = digits + 1
    return kept
```

The `digits` variable is there to track how much we can multiply the newest number by, so that it forms a coherent solution. Remember, everything exists for a reason.