Picturing a Function returning a Function

```python
def bigger_than ...
old = bigger_than(60)
print(old(65))
```

First, `bigger_than` is defined, which binds
that name to a function object (with nothing
yet filled in below the line: this function has
been definted but hasn’t been called yet).

Second, `bigger_than` is called with the
argument 60: it first binds `age` to 60 and
then defines the local function `test_it`:
which binds its name to a new function
object (this binding is shown below the line
in `bigger_than`).

Notice that the new function object `test_it`
stores a `closure`: it refers to a `dict` with the
name `age` bound to 60, the only binding
from the enclosing `bigger_than` function.

Then, `bigger_than` returns a reference to
`test_it`’s function object, which is bound to
the variable name `old` in the second
assignment statement.

Finally, the call to `bigger_than` disappears
(i.e., the information below the line
disappears), but the name `old` remains
bound to the function object originally
bound to `test_it` from when `bigger_than`
executed.

Third, `old` is called with the argument 65. It
binds `x` to 65 and then evaluates `x > age`
(with `x` bound to the argument 65 and `age`
bound to 60, from the closure). This
expression evaluates to `True`, which is
printed.
def bigger_than ...
old     = bigger_than(60)
ancient = bigger_than(90)
print(old(65),ancient(65))

Here is the result of calling bigger_than twice. Each call creates and returns a new function object with its own closure. The function object returned from the first call is bound to the name old, the function object returned from the second call is bound to the name ancient. In print, the first function call returns True and the second returns False.
When `f()` is called the first time, it returns references to the new `g1` and `g2` function objects created when `f` executes; these are bound to `f1` and `f2`. The enclosing scope of both of these function objects stores the name `prev`-local in the call to `f`- and its initial value is captured in the shared closure.

When `f()` is called the second time, it again returns references to the new `g1` and `g2` function objects created when `f` executes (new function objects declared in `f`); these in turn are bound to `f3` and `f4`. The enclosing scope of both of these function objects stores the name `prev`-local in the new call to `f`- and its initial value is captured in the shared closure.

Calling `f1(1)` stores into `temp` the value of `prev` in its function object’s closure (None), reassociates `prev` with 6, and returns `None`, which it prints in the console.

Calling `f2(2)` stores into `temp` the value of `prev` in its function object’s closure (the one shared with `f1`, now 6), reassociates that `prev` with 10, and returns 6, which it prints in the console.

Calling `f3(5)` stores into `temp` the value of `prev` in its function object’s closure (None), reassociates that `prev` with 10, and returns `None`, which it prints in the console.

Calling `f4(6)` stores into `temp` the value of `prev` in its function object’s closure (the one shared with `f1`, now 10), resets that `prev` to 30, and returns 10, which it prints in the console.