Scoping

almost always wrong.

```
def foo(xs=[]):
    ⋮
    xs.append(1)

foo() # xs = []
foo() # xs = [1]
```

length is undefined

```
def area(length, breadth = length):
    return length * breadth
```

Python workaround

```
def area(length, breadth = None):
    if breadth is None:
        breadth = length
    return length * breadth
```

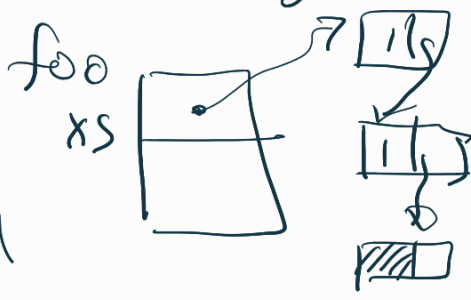
Object lifetimes ← when does list/int etc actually exist in memory.

Variable scope ~ which part of the Code can you use the name in?

When Python sees

```
def foo(xs=[]):
```

- Creates an empty list
- Records it as default value for first arg



Call `foo()`

- No first arg.
- Use the pointer recorded

Call `foo()`

- No first arg
- Use pointer recorded

```
int *foo() {
    int x = 1;
    return &x;
}
```

```
int *p = foo();
*p = 2;
```

} error

The object p refers to has died.

```
def make_ctr():
    v = 0
    def inc():
        nonlocal v
        v = v + 1
    def get():
        return v
    return (inc, get)
```

```
i, g = make_ctr()
i()
print(g())
i2, g2 = make_ctr()
i2()
i2()
print(g2())
```

Implement fns.

Fun (v, b)

- v → var
- b → AST of body

Call (e, f) ~ e(f) f

e, f → AST

$a(b)(c)$ — $a(b)$ is e, (c) is f

- eval e
- eval f
- add (v, f) to env
- eval b

Functions

a = f(x) + f(x)

↓ valid?

b = f(x)
a = b + b

Optimization:
Common sub-expression elimination

Some non-functions

```
y = 0
def foo(x):
    global y
    y = y + 1
    return x + 1
```

```
def foo(x):
    Print("Hello")
    return x + 1
```

→ dependency injection framework.

side-effects

Impure functions

function

```
def foo(x):
    return x+1
```

```
def fact(n)
    p = 1
    for i in range(2,n+1):
        p *= i
    return p
```

Pure functions

- Pure functions are easy to optimize.
- Pure functions are easy to test.

Functional Core + Imperative shell

Design pattern

```
def foo(x, printer = print):
    printer("Hello")
    return x+1
```