Attributes (of functions and classes and objects)

```
x.name
   ↑ attribute of x

x.name(parameters)
   ↑ also an attribute —
       happens to be function or method
```

Example

Suppose we want function f to have the equivalent of a "static variable" (remembered between calls). Can be done as follows:

```
def f(n):
    f.lastparam = n
    return 2*n
:
x = f(7)
print x, f.lastparam
```

Functions can also have "local" static variables,

~~[illegible]~~

default values

```
def f(n, y=[0], r=[1,2]):
    r[y[0]] = n
    y[0] = 1 - y[0]
    print r, y
    return n
```

Question can a function change (by assignment) a parameter passed to it?

```
def f(a):
    a = ' '.join(a)

a = ['this','is','a','test']
print f(a)
print a
```

what about

```
def f(a):
    a.reverse()

a = [1, 2, 3]
f(a)
print a
```

what's up with that?

Question what is the difference between the comparison "==" and "is"?

```
a = {'a': 2, 1: 0}
b = {1: 0, 'a': 2}
>>> a == b
>>> a is b
>>> c = a
>>> c is a
```

Classes

A convenient way to package functions that share some static variables
(Also has inheritance, instances, etc.)

```
class name:        # or  name (base...ladder ...)
    statements
```

Class attributes can be variables, functions, etc

```
class X:
    a = 0
    b = [17]
```

} executes immediately to define attributes

So, can write things like

```
X.b.append(6)
X.y = "sure"
print X.b
```

even before any object (instance of X) is constructed.

Methods

```
class X:
    def say(self):
        print self
```

Construction of instances (objects)

```
g = X()      <-- creates instance of class X
```

but what if you need parameters ?

```
g = X(a, b)
```

⇒ X needs to have __init__ method

```
class X:
    def __init__(self, s, t):
        self.v1 = s
        self.v2 = t
```

Now

```
g = X(1, 5)
print g.v1 , g.v2
```

Chapter 5 explains how classes and objects use dictionaries to associate attributes with their values