Framework for abstraction

		Object
primitives	+, *, ==	numbers, strings, True/False
Means of combination	if, while, … composition, e.g., can write 3*(4+7)	Data structures: lists dictionaries
Means of abstraction	def	Abstract data types
Means of capturing common patterns	higher-order procedures	Classes and inheritance

Computing square roots

```
def goodEnough(guess, x):
    return abs(x-square(guess)) < .0001

def sqrt(x):
  guess = 1.0
  while not goodEnough(guess, x):
    guess=average(guess, x/guess)
  return guess
```

```
def sqrt(x):
  def goodEnough(guess):
    def junk(y):
      …..
    return abs(x-square(guess)) < .0001
  guess = 1.0
  while not goodEnough(guess):
    guess=average(guess, x/guess)
  return guess
```

```
def frac(n,d):
   return [n,d]

def plus(s,t):
   return [s[0]*t[1] + s[1]*t[0], s[1]*t[1]]

>>> f1=frac(1,10)
>>> f2=frac(3,5)
>>> f1
[1, 10]
>>> f2
[3, 5]
>>> plus(f1,f2)
[35, 50]
```

Euclid's algorithm for computing greatest common divisors

```
def gcd(a,b):
   if b==0: return a
   else:
      return gcd(b, a % b)
```

```
def plus(s,t):
   n = s[0]*t[1] + s[1]*t[0]
   d = s[1]*t[1]
   g = gcd(n,d)
   return [n//g, d//g]

>>> f1
[1, 10]
>>> f2
[3, 5]
>>> plus(f1,f2)
[7, 10]
```

Abstract data type

- Specify data totally in terms of a set of operations that can be performed on it, **not** in terms of how the data is represented.
- Programmers manipulate the data **only** via these operations.

```
class frac:
    def __init__(self, n,d):
        g = gcd(n,d)
        self.numer=n//g
        self.denom=d//g

    def plus(self,other):
        return frac(
            (   self.numer  * other.denom   +
                self.denom  * other.numer),
            self.denom  * other.denom)

>>> a=frac(1,2)
>>> b=frac(3,4)
>>> c=frac.plus(a,b)
>>> c.numer
5
>>> c.denom
4
```

```
class frac:
    def __init__(self, n,d): …

    def plus(self, other): …

    def __repr__(self):
        return str(self.numer)+"/"+str(self.denom)


>>> frac(1,2)
1/2
>>>  frac.plus(frac(1,2),frac(3,4))
5/4
```

```
class frac:
    def __init__(self, n,d): …

    def plus(self, other): …

    def __repr__(self):  …

    def __add__(self, other):
        return frac.plus(self,other)


>>> frac(5,6)+frac(3,4)
19/12
```

Jargon to impress your friends

- Encapsulation
  - Internals are hidden and can be accessed only via well-defined application programming interfaces
- Polymorphism
  - The behavior of functions depends on the types of the arguments

```
class Account:
   def __init__(self,initialBalance):
      self.bal = initialBalance

   def balance(self):
      return self.bal

   def deposit(self,amount):
      self.bal = self.bal + amount

   def creditLimit(self):
      return self.balance() * 0.5
```

```
class PremierAccount:
   def __init__(self,initialBalance):
      self.bal = initialBalance

   def balance(self):
      return self.bal

   def deposit(self,amount):
      self.bal = self.bal + amount

   def creditLimit(self):
      return self.balance() * 1.5
```

```
class PremierAccount(Account):
   def creditLimit(self):
      return self.balance() * 1.5
```

Inheritance

PremierAccount → Account

Subclass Superclass

```
class FeeAccount(Account):
   def fee(self):
      return 1

   def deposit(self,amount):
      self.bal=self.bal + amount - self.fee()


>>>d=FeeAccount(100)
>>>d.fee()
1
>>>d.deposit(10)
>>>d.balance()
109
```

```
class MITAccount(PremierAccount, FeeAccount):
    pass

>>>m=MITAccount(100)
>>>m.fee()
1
>>>m.deposit(10)
>>>m.balance()
??
```

```
class TechAccount(FeeAccount, PremierAccount):
    pass

>>>t=TechAccount(100)
>>>t.fee()
1
>>>t.deposit(10)
>>>t.balance()
109
```