# Introduction

Brad Miller    David Ranum

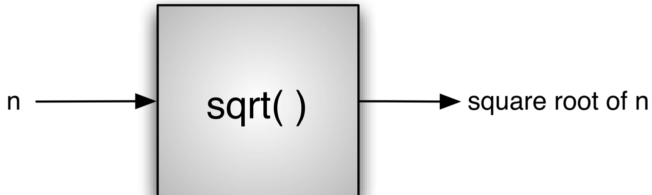
1/25/06

# Outline

1. Objectives
2. Getting Started
3. What Is Computer Science?
   - What Is Programming?
   - Why Study Data Structures and Abstract Data Types?
   - Why Study Algorithms?
4. Review of Basic Python
   - Getting Started with Data
   - Control Structures
   - Defining Functions
   - Object-Oriented Programming in Python: Defining Classes
5. Summary

- To review the ideas of computer science, programming, and problem-solving.
- To understand abstraction and the role it plays in the problem-solving process.
- To understand and implement the notion of an abstract data type.
- To review the Python programming language.

# Procedural Abstraction



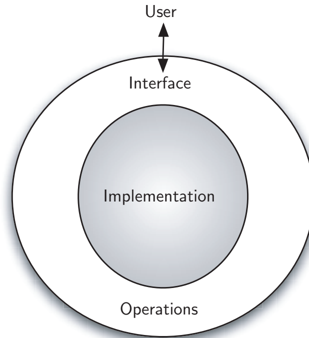
# Outline

1. Objectives
2. Getting Started
3. What Is Computer Science?
   - What Is Programming?
   - Why Study Data Structures and Abstract Data Types?
   - Why Study Algorithms?
4. Review of Basic Python
   - Getting Started with Data
   - Control Structures
   - Defining Functions
   - Object-Oriented Programming in Python: Defining Classes
5. Summary

# Outline

1. Objectives
2. Getting Started
3. What Is Computer Science?
   - What Is Programming?
   - Why Study Data Structures and Abstract Data Types?
   - Why Study Algorithms?
4. Review of Basic Python
   - Getting Started with Data
   - Control Structures
   - Defining Functions
   - Object-Oriented Programming in Python: Defining Classes
5. Summary

# Abstract Data Type



# Outline

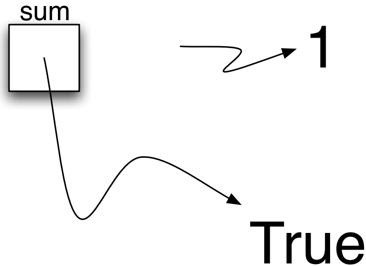
1. Objectives
2. Getting Started
3. What Is Computer Science?
   - What Is Programming?
   - Why Study Data Structures and Abstract Data Types?
   - Why Study Algorithms?
4. Review of Basic Python
   - Getting Started with Data
   - Control Structures
   - Defining Functions
   - Object-Oriented Programming in Python: Defining Classes
5. Summary

# Outline

1. Objectives
2. Getting Started
3. What Is Computer Science?
   - What Is Programming?
   - Why Study Data Structures and Abstract Data Types?
   - Why Study Algorithms?
4. Review of Basic Python
   - Getting Started with Data
   - Control Structures
   - Defining Functions
   - Object-Oriented Programming in Python: Defining Classes
5. Summary

# Variables Hold References to Data Objects

# Assignment Changes the Reference



# Outline

1. Objectives
2. Getting Started
3. What Is Computer Science?
   - What Is Programming?
   - Why Study Data Structures and Abstract Data Types?
   - Why Study Algorithms?
4. Review of Basic Python
   - Getting Started with Data
   - Control Structures
   - Defining Functions
   - Object-Oriented Programming in Python: Defining Classes
5. Summary

# Outline

1. Objectives
2. Getting Started
3. What Is Computer Science?
   - What Is Programming?
   - Why Study Data Structures and Abstract Data Types?
   - Why Study Algorithms?
4. Review of Basic Python
   - Getting Started with Data
   - Control Structures
   - Defining Functions
   - Object-Oriented Programming in Python: Defining Classes
5. Summary

# Function to Compute a Square Root Using Newton's Method

```
1  def squareroot(n):
2      root = n/2
3      for k in range(20):
4          root = (1.0/2)*(root + (n / root))
5
6      return root
```

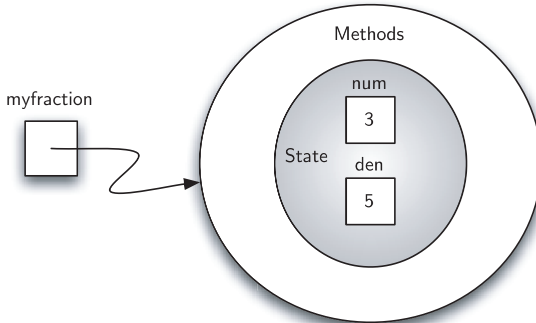
# Outline

1. Objectives
2. Getting Started
3. What Is Computer Science?
   - What Is Programming?
   - Why Study Data Structures and Abstract Data Types?
   - Why Study Algorithms?
4. Review of Basic Python
   - Getting Started with Data
   - Control Structures
   - Defining Functions
   - Object-Oriented Programming in Python: Defining Classes
5. Summary

# Fraction Class with the Constructor

```
1  class Fraction:
2
3      def __init__(self,top,bottom):
4
5          self.num = top
6          self.den = bottom
```

# An Instance of the `Fraction` Class



# show Method for Fractions

```
1     def show(self):
2         print self.num,"/",self.den
```

# `__str__` Method for Fractions

```
1       def __str__(self):
2           return str(self.num)+"/"+str(self.den)
```

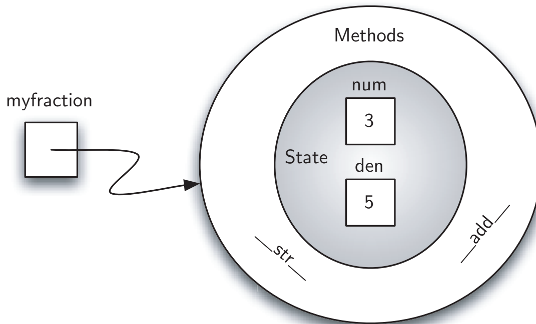
# __add__ Method for Fractions

```
1      def __add__(self,otherfraction):
2
3          newnum = self.num*otherfraction.den + \
4                       self.den*otherfraction.num
5          newden = self.den * otherfraction.den
6
7          return Fraction(newnum,newden)
```

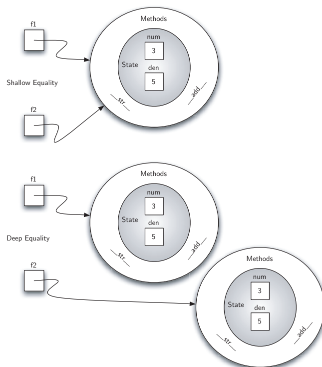
# Greatest Common Divisor Function

```
1  #Assume that m and n are greater than zero
2  def gcd(m,n):
3      while m%n != 0:
4          oldm = m
5          oldn = n
6
7          m = oldn
8          n = oldm%oldn
9
10     return n
```

# An Instance of the `Fraction` Class with Two Methods



# Shallow Equality Versus Deep Equality



# __cmp__ Method for Fractions

```
1      def __cmp__(self,otherfraction):
2
3          num1 = self.num*otherfraction.den
4          num2 = self.den*otherfraction.num
5
6          if num1 < num2:
7              return -1
8          else:
9              if num1 == num2:
10                 return 0
11             else
12                 return 1
```

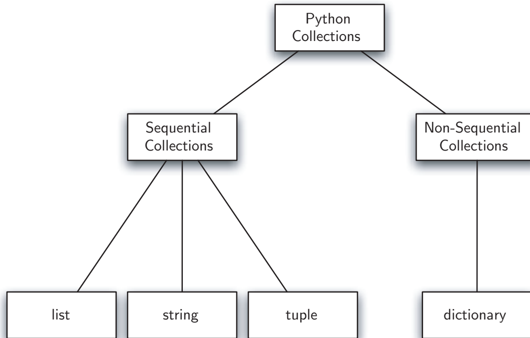
# Fraction Class I

```
1  class Fraction:
2      def __init__(self,top,bottom):
3          self.num = top
4          self.den = bottom
5
6      def __str__(self):
7          return str(self.num)+"/"+str(self.den)
8
9      def show(self):
10         print self.num,"/",self.den
11
12     def __add__(self,otherfraction):
13         newnum = self.num*otherfraction.den + \
14                      self.den*otherfraction.num
15         newden = self.den * otherfraction.den
```

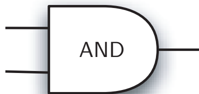
# Fraction Class II

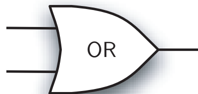
```
16          common = gcd(newnum,newden)
17          return Fraction(newnum/common,newden/common)
18
19      def __cmp__(self,otherfraction):
20          num1 = self.num*otherfraction.den
21          num2 = self.den*otherfraction.num
22          if num1 < num2:
23             return -1
24         else:
25             if num1 == num2:
26                return 0
27             else:
28                return 1
```

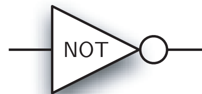
# An Inheritance Hierarchy for Python Collections



# Three Types of Logic Gates



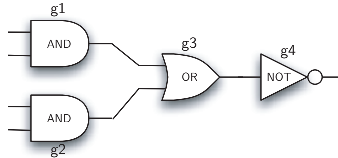



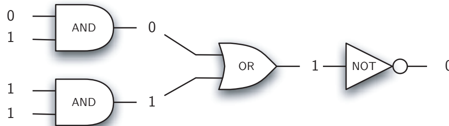

| and | 0 | 1 |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |

| or | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 1 |

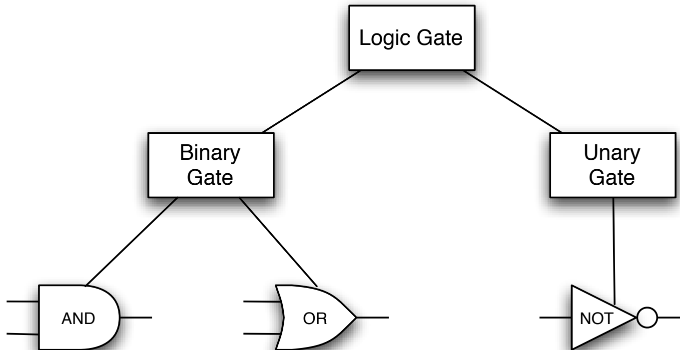
| not | |
|---|---|
| 0 | 1 |
| 1 | 0 |

# Circuit



# An Inheritance Hierarchy for Logic Gates



## Superclass LogicGate

```
1  class LogicGate:
2
3      def __init__(self,n):
4          self.label = n
5          self.output = None
6
7      def getLabel(self):
8          return self.label
9
10     def getOutput(self):
11         self.output = self.performGateLogic()
12         return self.output
```

# The BinaryGate Class

```
1   class BinaryGate(LogicGate):
2
3       def __init__(self,n):
4           LogicGate.__init__(self,n)
5
6           self.pinA = None
7           self.pinB = None
8
9       def getPinA(self):
10          return input("Enter Pin A input for gate "+ \
11                              self.getLabel()+"-->")
12
13      def getPinB(self):
14          return input("Enter Pin B input for gate "+ \
15                              self.getLabel()+"-->")
```

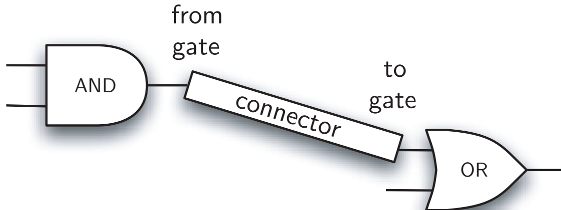
# The UnaryGate Class

```
1  class UnaryGate(LogicGate):
2
3      def __init__(self,n):
4          LogicGate.__init__(self,n)
5
6          self.pin = None
7
8      def getPin(self):
9          return input("Enter Pin input for gate "+ \
10                                 self.getLabel()+"-->")
```

# The AndGate Class

```
1  class AndGate(BinaryGate):
2
3      def __init__(self,n):
4          BinaryGate.__init__(self,n)
5
6      def performGateLogic(self):
7
8          a = self.getPinA()
9          b = self.getPinB()
10         if a==1 and b==1:
11             return 1
12         else:
13             return 0
```

# A Connector Connects the Output of One Gate to the Input of Another



# The Connector Class

```
class Connector:

    def __init__(self, fgate, tgate):
        self.fromgate = fgate
        self.togate = tgate

        tgate.setNextPin(self)

    def getFrom(self):
        return self.fromgate

    def getTo(self):
        return self.togate
```

# The setNextPin Method

```
1     def setNextPin(self,source):
2         if self.pinA == None:
3             self.pinA = source
4         else:
5             if self.pinB == None:
6                 self.pinB = source
7             else:
8                 print "Cannot Connect: NO EMPTY PINS"
```

# A Modified getPin Method

```
1        def getPinA(self):
2            if self.pinA == None:
3                return input("Enter Pin A input for gate "+ \
4                                self.getName()+"-->")
5            else:
6                return self.pinA.getFrom().getOutput()
```

# The Circuit Classes I

```
1  class LogicGate:
2
3      def __init__(self,n):
4          self.label = n
5          self.output = None
6
7      def getLabel(self):
8          return self.label
9
10     def getOutput(self):
11         self.output = self.performGateLogic()
12         return self.output
13
14
15
```

# The Circuit Classes II

```
16
17   class BinaryGate(LogicGate):
18
19       def __init__(self,n):
20           LogicGate.__init__(self,n)
21
22           self.pinA = None
23           self.pinB = None
24
25       def getPinA(self):
26           if self.pinA == None:
27               return input("Enter Pin A input for gate "+ \
28                                   self.getLabel()+"-->")
29           else:
30               return self.pinA.getFrom().getOutput()
31
```

# The Circuit Classes III

```
32        def getPinB(self):
33            if self.pinB == None:
34                return input("Enter Pin B input for gate "+ \
35                                     self.getLabel()+"-->")
36            else:
37                return self.pinB.getFrom().getOutput()
38
39        def setNextPin(self,source):
40             if self.pinA == None:
41                 self.pinA = source
42             else:
43                 if self.pinB == None:
44                     self.pinB = source
45                 else:
46                     print "Cannot Connect: NO EMPTY PINS"
47
```

# The Circuit Classes IV

```
48  class AndGate(BinaryGate):
49
50      def __init__(self,n):
51          BinaryGate.__init__(self,n)
52
53      def performGateLogic(self):
54          a = self.getPinA()
55          b = self.getPinB()
56          if a==1 and b==1:
57              return 1
58          else:
59              return 0
60
61
62
63
```

# The Circuit Classes V

```
64 class OrGate(BinaryGate):
65
66     def __init__(self,n):
67         BinaryGate.__init__(self,n)
68
69     def performGateLogic(self):
70         a = self.getPinA()
71         b = self.getPinB()
72         if a ==1 or b==1:
73             return 1
74         else:
75             return 0
76
77
78
79
```

# The Circuit Classes VI

```
80
81  class UnaryGate(LogicGate):
82
83      def __init__(self,n):
84          LogicGate.__init__(self,n)
85
86          self.pin = None
87
88      def getPin(self):
89          if self.pin == None:
90              return input("Enter Pin input for gate "+ \
91                                  self.getLabel()+"-->")
92          else:
93              return self.pin.getFrom().getOutput()
94
95      def setNextPin(self,source):
```

# The Circuit Classes VII

```
96              if self.pin == None:
97               self.pin = source
98          else:
99               print "Cannot Connect: NO EMPTY PINS"
100
101 class NotGate(UnaryGate):
102
103     def __init__(self,n):
104         UnaryGate.__init__(self,n)
105
106     def performGateLogic(self):
107         if self.getPin():
108             return 0
109         else:
110          return 1
111
```

# The Circuit Classes VIII

```
112 class Connector:
113
114     def __init__(self, fgate, tgate):
115         self.fromgate = fgate
116         self.togate = tgate
117
118         tgate.setNextPin(self)
119
120     def getFrom(self):
121         return self.fromgate
122
123     def getTo(self):
124         return self.togate
```

- Computer science is the study of problem-solving.
- Computer science uses abstraction as a tool for representing both processes and data.
- Abstract data types allow programmers to manage the complexity of a problem domain by hiding the details of the data.
- Python is a powerful, yet easy-to-use, object-oriented language.

- Lists, tuples, and strings are built in Python sequential collections.
- Dictionaries are nonsequential collections of data.
- Classes allow programmers to implement abstract data types.
- Programmers can override standard methods as well as create new methods.
- Classes can be organized into hierarchies.
- A class constructor should always invoke the constructor of its parent before continuing on with its own data and behavior.