Primitives, Combination, Abstraction, and Patterns
Python has features that facilitate modular programming.

- **def** combines operations into a procedure and binds a name to it
- **lists** provide flexible and hierarchical structures for data
- **variables** associate names with data
- **classes** associate data (attributes) and procedures (methods)

<table>
<thead>
<tr>
<th>Procedures</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Primitives</strong></td>
<td>+, *, ==, !=</td>
</tr>
<tr>
<td><strong>Combination</strong></td>
<td>if, while, f(g(x))</td>
</tr>
<tr>
<td><strong>Abstraction</strong></td>
<td>def</td>
</tr>
<tr>
<td><strong>Patterns</strong></td>
<td>higher-order procedures</td>
</tr>
<tr>
<td></td>
<td>numbers, booleans, strings</td>
</tr>
<tr>
<td></td>
<td>lists, dictionaries, objects</td>
</tr>
<tr>
<td></td>
<td>classes</td>
</tr>
<tr>
<td></td>
<td>super-classes, sub-classes</td>
</tr>
</tbody>
</table>
PCAP Framework for Managing Complexity

We will build on these ideas to manage complexity at higher levels.

- **Programming Styles** for dealing with complexity
- PCAP in Higher-Level Abstractions: **State Machines**

**Reading:** Course notes, chapters 3–4
Programming Styles for Managing Complexity

Structure of program has significant effect on its modularity.

**Imperative** (procedural) programming

- focus on step-by-step instructions to accomplish task
- organize program using structured conditionals and loops

**Functional** programming

- focus on procedures that mimic mathematical functions, producing outputs from inputs without side effects
- functions are *first-class objects* used in data structures, arguments to procedures, and can be returned by procedures

**Object-oriented** programming

- focus on collections of related procedures and data
- organize programs as hierarchies of related classes and instances
Example Program

Task: Find a sequence of operations (either increment or square) that transforms the integer \(i\) (initial) to the integer \(g\) (goal).

Example: applying the sequence

\[
\text{increment increment increment square}
\]

to 1 yields \(16\)

apply increment to 1 \(\rightarrow 2\)
apply increment to 2 \(\rightarrow 3\)
apply increment to 3 \(\rightarrow 4\)
apply square to 4 \(\rightarrow 16\)
What is the minimum length sequence of **increment** and **square** operations needed to transform 1 to 100?

1: <4  
2: 4  
3: 5  
4: 6  
5: >6
Check Yourself

What is the minimum length sequence of increment and square operations needed to transform 1 to 100?

Try to use as many squares (especially big ones) as possible.

apply increment to 1 → 2
apply increment to 2 → 3
apply square to 3 → 9
apply increment to 9 → 10
apply square to 10 → 100

Five operations.
Check Yourself

What is the minimum length sequence of increment and square operations needed to transform 1 to 100? 3: 5

1: <4  2: 4  3: 5  4: 6  5: >6
Solve the previous problem by writing an imperative program to step through all possible sequences of length 1, 2, 3, ...

```python
def increment(n):
    return n+1

def square(n):
    return n**2

def findSequence(initial,goal):
    # construct list of "candidates" of form ('1 increment increment',3)
    candidates = [(str(initial),initial)]
    # loop over sequences of length "i" = 1, 2, 3, ...
    for i in range(1,goal-initial+1):
        newCandidates = []
        # construct each new candidate by adding one operation to prev candidate
        for (action,result) in candidates:
            for (a,r) in [('increment',increment),('square',square)]:
                newCandidates.append((action+a,r(result)))
        print i,':',newCandidates[-1]
        if newCandidates[-1][1] == goal:
            return newCandidates[-1]
        candidates = newCandidates

answer = findSequence(1,100)
print 'answer =',answer
```
Imperative (Procedural) Programming

1: ('1 increment', 2)
1: ('1 square', 1)
2: ('1 increment increment', 3)
2: ('1 increment square', 4)
2: ('1 square increment', 2)
2: ('1 square square', 1)
3: ('1 increment increment increment', 4)
3: ('1 increment increment square', 9)
3: ('1 increment square increment', 5)
3: ('1 increment square square', 16)
3: ('1 square increment increment', 3)
3: ('1 square increment square', 4)
3: ('1 square square increment', 2)
3: ('1 square square square', 1)
4: ('1 increment increment increment increment', 5)
4: ('1 increment increment increment square', 16)
4: ('1 increment increment square increment', 10)
4: ('1 increment increment square square', 81)
4: ('1 increment square increment increment', 6)
4: ('1 increment square increment square', 25)
4: ('1 increment square square increment', 17)
4: ('1 increment square square square', 256)
4: ('1 square increment increment increment', 4)
4: ('1 square increment increment square', 9)
4 : ('1 square increment square increment', 5)
4 : ('1 square increment square square', 16)
4 : ('1 square square increment increment', 3)
4 : ('1 square square increment square', 4)
4 : ('1 square square square increment', 2)
4 : ('1 square square square square', 1)
5 : ('1 increment increment increment increment increment increment', 6)
5 : ('1 increment increment increment increment increment square', 25)
5 : ('1 increment increment increment increment increment square increment', 17)
5 : ('1 increment increment increment increment square square', 256)
5 : ('1 increment increment increment square increment increment', 11)
5 : ('1 increment increment square increment square square', 100)

answer = ('1 increment increment square increment square square', 100)
Imperative (Procedural) Programming

This imperative version of the program has three levels of looping.

```python
def findSequence(initial, goal):
    # construct list of "candidates" of form ('1 increment increment',3)
    candidates = [(str(initial), initial)]
    # loop over sequences of length "i" = 1, 2, 3, ...
    for i in range(1, goal-initial+1):
        newCandidates = []
        # construct each new candidate by adding one operation to prev candidate
        for (action, result) in candidates:
            for (a, r) in [('increment', increment), ('square', square)]:
                newCandiates.append((action+a, r(result)))
            print i, ': ', newCandidates[-1]
            if newCandidates[-1][1] == goal:
                return newCandidates[-1]
        candidates = newCandidates
```

This approach is straightforward, but nested loops can be confusing.

Challenge is to get the indices right.
Functional Programming

This version focuses on functions as primitives.

```python
def apply(opList, arg):
    if len(opList) == 0:
        return arg
    else:
        return apply(opList[1:], opList[0](arg))

def addLevel(opList, fctList):
    return [x+y for y in fctList for x in opList]

def findSequence(initial, goal):
    opList = [[]]
    for i in range(1, goal-initial+1):
        opList = addLevel(opList, [increment, square])
        for seq in opList:
            if apply(seq, initial) == goal:
                return seq

answer = findSequence(1, 100)
print 'answer =', answer
```
The procedure \texttt{apply} is a “pure function.”

\begin{verbatim}
def apply(opList, arg):
    if len(opList) == 0:
        return arg
    else:
        return apply(opList[1:], opList[0](arg))
\end{verbatim}

Its first argument is a list of functions. The procedure applies these functions to the second argument \texttt{arg} and returns the result.

\begin{verbatim}
>>> apply([], 7)
7
>>> apply([increment], 7)
8
>>> apply([square], 7)
49
>>> apply([increment, square], 7)
64
\end{verbatim}

This list of procedures uses functions as first-class objects.
The procedure `addLevel` is also a pure function.

```python
def addLevel(opList,fctList):
    return [x+[y] for y in fctList for x in opList]
```

The first input is a list of sequences-of-operations, each of which is a list of functions.

The second input is a list of possible next-functions.

It returns a new list of sequences.

```python
>>> addLevel([['increment']],[['increment','square']])
[['<function increment at 0xb7480aac>, <function increment at 0xb7480aac>'],
[['function increment at 0xb7480aac>, <function square at 0xb747b25c>]]
```
The answer is now a list of functions.

def apply(opList, arg):
    if len(opList) == 0:
        return arg
    else:
        return apply(opList[1:], opList[0](arg))

def addLevel(opList, fctList):
    return [x + [y] for y in fctList for x in opList]

def findSequence(initial, goal):
    opList = [[]]
    for i in range(1, goal - initial + 1):
        opList = addLevel(opList, [increment, square])
        for seq in opList:
            if apply(seq, initial) == goal:
                return seq

answer = findSequence(1, 100)
print 'answer =', answer

answer = [<function increment at 0xb777ea74>, <function increment at 0xb777ea74>, <function square at 0xb7779224>, <function increment at 0xb777ea74>, <function square at 0xb7779224>]
The functions **apply** and **addLevel** are easy to check.

```python
def apply(opList, arg):
    if len(opList) == 0:
        return arg
    else:
        return apply(opList[1:], opList[0](arg))

def addLevel(opList, fctList):
    return [x+[y] for y in fctList for x in opList]
```

```console
>>> apply([], 7)
7
>>> apply([increment], 7)
8
>>> apply([square], 7)
49
>>> apply([increment, square], 7)
64
>>> addLevel([[increment]], [increment, square])
[[<function increment at 0xb7480aac>, <function increment at 0xb7480aac>],
 [<function increment at 0xb7480aac>, <function square at 0xb747b25c>]]
```

Greater modularity reduces complexity and simplifies debugging.
Functional Programming

Also notice that the definition of `apply` is recursive: the definition of `apply` calls `apply`.

```python
>>> def apply(oplist, arg):
...     if len(opList) == 0:
...         return arg
...     else:
...         return apply(opList[1:], opList[0](arg))
```

Recursion is

- an alternative way to implement iteration (looping)
- a natural generalization of functional programming
- powerful way to think about PCAP
Recursion

Express solution to problem in terms of simpler version of problem.

Example: raising a number to a non-negative integer power

\[ b^n = \begin{cases} 
1 & \text{if } n = 0 \\
 b \cdot b^{n-1} & \text{if } n > 0 
\end{cases} \]

functional notation:

\[ f(n) = \begin{cases} 
1 & \text{if } n = 0 \\
 b f(n - 1) & \text{if } n > 0 
\end{cases} \]

Python implementation:

```python
def exponent(b, n):
    if n == 0:
        return 1
    else:
        return b * exponent(b, n-1)
```
Recursive Exponentiation

Invoking \textbf{exponent}(2, 6) generates 6 more invocations of \textbf{exponent}.

\begin{verbatim}
def exponent(b,n):
    if n==0:
        return 1
    else:
        return b*exponent(b,n-1)

exponent(2,6)
calls exponent(2,5)
calls exponent(2,4)
calls exponent(2,3)
calls exponent(2,2)
calls exponent(2,1)
calls exponent(2,0)
returns 1
returns 2
returns 4
returns 8
returns 16
returns 32
returns 64
64
\end{verbatim}

Number of invocations increases in proportion to \textbf{n} (i.e., linearly).
Fast Exponentiation

There is a straightforward way to speed this process:
If $n$ is even, then square the result of raising $b$ to the $n/2$ power.

$$b^n = \begin{cases} 
1 & \text{if } n = 0 \\
 b \cdot b^{n-1} & \text{if } n \text{ odd} \\
(b^{n/2})^2 & \text{otherwise}
\end{cases}$$

functional notation:

$$f(n) = \begin{cases} 
1 & \text{if } n = 0 \\
 bf(n-1) & \text{if } n \text{ odd} \\
(f(n/2))^2 & \text{otherwise}
\end{cases}$$
Implement in Python.

def fastExponent(b, n):
    if n == 0:
        return 1
    elif n % 2 == 1:
        return b * fastExponent(b, n-1)
    else:
        return fastExponent(b, n/2)**2
def fastExponent(b,n):
    if n==0:
        return 1
    elif n%2==1:
        return b*fastExponent(b,n-1)
    else:
        return fastExponent(b,n/2)**2

How many invocations of \texttt{fastExponent} is generated by \texttt{fastExponent (2,10)}? 

1. 10 
2. 8 
3. 7 
4. 6 
5. 5
Recursive Exponentiation

Implement recursion in Python.

def fastExponent(b,n):
    if n==0:
        return 1
    elif n%2==1:
        return b*fastExponent(b,n-1)
    else:
        return fastExponent(b,n/2)**2

fastExponent(2,10)
calls fastExponent(2,5)
calls fastExponent(2,4)
calls fastExponent(2,2)
calls fastExponent(2,1)
calls fastExponent(2,0)
returns 1
returns 2
returns 4
returns 16
returns 32
returns 1024
1024

The number of calls increases in proportion to \( \log n \) (for large \( n \)).
def fastExponent(b, n):
    if n == 0:
        return 1
    elif n % 2 == 1:
        return b * fastExponent(b, n-1)
    else:
        return fastExponent(b, n/2)**2

How many invocations of `fastExponent` is generated by `fastExponent(2, 10)`?

<table>
<thead>
<tr>
<th>Option</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>10</td>
</tr>
<tr>
<td>2.</td>
<td>8</td>
</tr>
<tr>
<td>3.</td>
<td>7</td>
</tr>
<tr>
<td>4.</td>
<td>6</td>
</tr>
<tr>
<td>5.</td>
<td>5</td>
</tr>
</tbody>
</table>
Recursive Exponentiation

Functional approach makes this simplification easy to spot.

def fastExponent(b,n):
    if n==0:
        return 1
    elif n%2==1:
        return b*fastExponent(b,n-1)
    else:
        return fastExponent(b,n/2)**2

fastExponent(2,10)
calls fastExponent(2,5)
calls fastExponent(2,4)
calls fastExponent(2,2)
calls fastExponent(2,1)
calls fastExponent(2,0)
returns 1
returns 2
returns 4
returns 16
returns 32
returns 1024
1024

Functional approach is “expressive.”
Towers of Hanoi

Transfer a stack of disks from post A to post B by moving the disks one-at-a-time, without placing any disk on a smaller disk.

def Hanoi(n, A, B, C):
    if n==1:
        print 'move from ' + A + ' to ' + B
    else:
        Hanoi(n-1, A, C, B)
        Hanoi(1, A, B, C)
        Hanoi(n-1, C, B, A)
Towers of Hanoi

Towers of height 3 and 4.

> > > Hanoi(3,'a','b','c')
move from a to b
move from a to c
move from b to c
move from a to b
move from c to a
move from c to b
move from a to b

> > > Hanoi(4,'a','b','c')
move from a to c
move from a to b
move from c to b
move from a to c
move from b to a
move from b to c
move from a to c
move from a to b
move from c to b
move from c to a
move from b to a
move from c to b
move from a to c
move from a to b
move from c to b
move from a to c
move from b to a
move from c to b
move from a to b
move from c to b
Towers of Hanoi

Transfer a stack of disks from post A to post B by moving the disks one-at-a-time, without placing any disk on a smaller disk.

```
def Hanoi(n,A,B,C):
    if n==1:
        print 'move from ' + A + ' to ' + B
    else:
        Hanoi(n-1,A,C,B)
        Hanoi(1,A,B,C)
        Hanoi(n-1,C,B,A)
```

Recursive solution is “expressive” (also simple and elegant).
Back to the Earlier Example

Task: Find a sequence of operations (either increment or square) that transforms the integer \( i \) (initial) to the integer \( g \) (goal).

**Imperative** (procedural) approach ✓

**Functional** approach ✓

**Object-oriented** approach
Represent all possible sequences in a tree.

Define an object to represent each of these “nodes”:

```python
class Node:
    def __init__(self, parent, action, answer):
        self.parent = parent
        self.action = action
        self.answer = answer

    def path(self):
        if self.parent == None:
            return [(self.action, self.answer)]
        else:
            return self.parent.path() + [(self.action, self.answer)]
```
Systematically create and search through all possible **Nodes**

```python
def findSequence(initial, goal):
    q = [Node(None, None, 1)]
    while q:
        parent = q.pop(0)
        for (a, r) in [('increment', increment), ('square', square)]:
            newNode = Node(parent, a, r(parent.answer))
            if newNode.answer == goal:
                return newNode.path()
            else:
                q.append(newNode)
    return None

answer = findSequence(1, 100)
print('answer =', answer)

answer = [(None, 1), ('increment', 2), ('increment', 3), ('square', 9), ('increment', 10), ('square', 100)]
```

Focus on constructing objects that represent pieces of the solution. More later, when we focus on effective **search** strategies.
Task: Find a sequence of operations (either increment or square) that transforms the integer \( i \) (initial) to the integer \( g \) (goal).

**Imperative** (procedural) approach
- structure of search was embedded in loops

**Functional** approach
- structure of search was constructed in lists of functions

**Object-oriented** approach
- structure of search was constructed from objects

Structure of program has significant effect on its modularity.

Now consider abstractions at even higher levels.
Controlling Processes

Programs that control the evolution of processes are different.

Examples:
  • bank accounts
  • graphical user interfaces
  • controllers (robotic steering)

We need a different kind of abstraction.
State Machines

Organizing computations that evolve with time.

On the $n^{th}$ step, the system
- gets input $i_n$
- generates output $o_n$ and
- moves to a new state $s_{n+1}$

Output and next state depend on input and current state

Explicit representation of stepwise nature of required computation.
State Machines

Example: Turnstile

Inputs = \{\text{coin, turn, none}\}

Outputs = \{\text{enter, pay}\}

States = \{\text{locked, unlocked}\}

\[
\text{nextState}(s, i) = \begin{cases} 
\text{unlocked} & \text{if } i = \text{coin} \\
\text{locked} & \text{if } i = \text{turn} \\
s & \text{otherwise}
\end{cases}
\]

\[
\text{output}(s, i) = \begin{cases} 
\text{enter} & \text{if } \text{nextState}(s, i) = \text{unlocked} \\
\text{pay} & \text{otherwise}
\end{cases}
\]

\[s_0 = \text{locked}\]
State-transition Diagram

Graphical representation of process.

- Nodes represent states
- Arcs represent transitions: label is input / output

Locked

Unlock

Start

None/pay

Turn/pay

Coin/enter

Coin/enter

Turn/pay

None/enter

None/enter
Turn Table

Transition table.

<table>
<thead>
<tr>
<th>time</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>state</td>
<td>locked</td>
<td>locked</td>
<td>unlocked</td>
<td>unlocked</td>
<td>locked</td>
<td>locked</td>
<td>unlocked</td>
</tr>
<tr>
<td>input</td>
<td>none</td>
<td>coin</td>
<td>none</td>
<td>turn</td>
<td>coin</td>
<td>coin</td>
<td>coin</td>
</tr>
<tr>
<td>output</td>
<td>pay</td>
<td>enter</td>
<td>enter</td>
<td>pay</td>
<td>enter</td>
<td>enter</td>
<td>enter</td>
</tr>
</tbody>
</table>
State Machines

The state machine representation for controlling processes

- is simple and concise
- separates system specification from looping structures over time
- is modular

We will use this approach in controlling our robots.
Modular Design with State Machines

Break complicated problems into parts.

Example: consider exploration with mapping
Modular Design with State Machines

Break complicated problems into parts.

Map: black and red parts.
Plan: blue path, with **heading** determined by first line segment.
State Machines in Python

Represent common features of all state machines in the SM class. Represent kinds of state machines as subclasses of SM. Represent particular state machines as instances.

Example of hierarchical structure

SM Class: All state machines share some methods:
- start(self)  — initialize the instance
- step(self, input)  — receive and process new input
- transduce(self, inputs)  — make repeated calls to step

Turnstile Class: All turnstiles share some methods and attributes:
- startState  — initial contents of state
- getNextValues(self, state, inp)  — method to process input

Turnstile Instance: Attributes of this particular turnstile:
- state  — current state of this turnstile
The generic methods of the **SM** class use `startState` to initialize the instance variable `state`. Then `getNextValues` is used to process inputs, so that `step` can update `state`.

```python
class SM:
    def start(self):
        self.state = self.startState

    def step(self, inp):
        (s, o) = self.getNextValues(self.state, inp)
        self.state = s
        return o

    def transduce(self, inputs):
        self.start()
        return [self.step(inp) for inp in inputs]
```

Note that `getNextValues` should not change `state`. The `state` is managed by `start` and `step`. 
All turnstiles share the same `startState` and `getNextValues`.

class Turnstile(SM):
    startState = 'locked'

    def getNextValues(self, state, inp):
        if inp == 'coin':
            return ('unlocked', 'enter')
        elif inp == 'turn':
            return ('locked', 'pay')
        elif state == 'locked':
            return ('locked', 'pay')
        else:
            return ('unlocked', 'enter')
**Turn, Turn, Turn**

A particular turnstyle `ts` is represented by an instance.

```python
testInput = [None, 'coin', None, 'turn', 'turn', 'coin', 'coin']
ts = Turnstile()
ts.transduce(testInput)
```

Start state: locked

<table>
<thead>
<tr>
<th>In</th>
<th>Out</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>pay</td>
<td>locked</td>
</tr>
<tr>
<td>coin</td>
<td>enter</td>
<td>unlocked</td>
</tr>
<tr>
<td>None</td>
<td>enter</td>
<td>unlocked</td>
</tr>
<tr>
<td>turn</td>
<td>pay</td>
<td>locked</td>
</tr>
<tr>
<td>turn</td>
<td>pay</td>
<td>locked</td>
</tr>
<tr>
<td>coin</td>
<td>enter</td>
<td>unlocked</td>
</tr>
<tr>
<td>coin</td>
<td>enter</td>
<td>unlocked</td>
</tr>
</tbody>
</table>

`['pay', 'enter', 'enter', 'pay', 'pay', 'enter', 'enter']`
class Accumulator(SM):
    startState = 0

    def getNextValues(self, state, inp):
        return (state + inp, state + inp)
>>> a = Accumulator()
>>> a.start()
>>> a.step(7)
>>> b = Accumulator()
>>> b.start()
>>> b.step(10)
>>> a.step(-2)
>>> print a.state,a.getNextValues(8,13),b.getNextValues(8,13)

What will be printed?

1:  5 (18, 18) (23, 23)
2:  5 (21, 21) (21, 21)
3:  15 (18, 18) (23, 23)
4:  15 (21, 21) (21, 21)
5: none of the above
Classes and Instances for Accumulator

```
a = Accumulator()
a.start()
a.step(7)
b = Accumulator()
b.start()
b.step(10)
a.step(-2)
```
>>> a = Accumulator()
>>> a.start()
>>> a.step(7)
>>> b = Accumulator()
>>> b.start()
>>> b.step(10)
>>> a.step(-2)
>>> print a.state, a.getNextValues(8, 13), b.getNextValues(8, 13)

What will be printed? 2

1: 5 (18, 18) (23, 23)
2: 5 (21, 21) (21, 21)
3: 15 (18, 18) (23, 23)
4: 15 (21, 21) (21, 21)
5: none of the above
State Machine Combinators

State machines can be **combined** for more complicated tasks.

- **Cascade** ($M_1, M_2$)
  - Input: $i_1$
  - Output: $o_1 = i_2$

- **Parallel** ($M_1, M_2$)
  - Input: $i_1, i_2$
  - Output: $o_1, o_2$

- **Feedback** ($M_1$)
  - Input: $i_1[0], i_1[1]$
  - Output: $o_1$
```python
>>> a = Accumulator()
>>> b = Accumulator()
>>> c = Cascade(a,b)
>>> print c.transduce([7,3,4])
```

What will be printed?

1: [7, 3, 4]
2: [7, 10, 14]
3: [7, 17, 31]
4: [0, 7, 17]
5: none of the above
Check Yourself

```python
>>> a = Accumulator()
>>> b = Accumulator()
>>> c = Cascade(a, b)
>>> print c.transduce([7,3,4])
```

![Diagram](image-url)
Check Yourself

```python
>>> a = Accumulator()
>>> b = Accumulator()
>>> c = Cascade(a,b)
>>> print c.transduce([7,3,4])
```

What will be printed? 3

1: [7, 3, 4]
2: [7, 10, 14]
3: [7, 17, 31]
4: [0, 7, 17]
5: none of the above
This Week

Software lab: Practice with simple state machines

Design lab: Controlling robots with state machines

Homework 1: Symbolic calculator