**Finding the Error**

- Unit tests cannot find the source of an error
- Idea: “Visualize” the program with print statements

```python
def last_name_first(n):
    """Returns: copy of <n> in form <last>, <first>""
    end_first = n.find(' ')
    print end_first
    first = n[:end_first]
    print 'first is ' + str(first)
    last = n[end_first+1:]
    print 'last is ' + str(last)
    return last + ', ' + first
```

**Structure vs. Flow**

**Program Structure**
- Way statements are presented
  - Order statements are listed
  - Inside/outside of a function
  - Will see other ways...

**Program Flow**
- Order statements are executed
  - Not the same as structure
  - Some statements duplicated
  - Some statements are skipped
  - Indicate possibilities over multiple executions
  - Indicates what really happens in a single execution

**Structure vs. Flow: Example**

**Program Structure**
```python
def foo():
    print 'Hello'
    # Application code
    if __name__ == 'main':
        foo()
        foo()
        foo()
```

**Program Flow**
```bash
>>> python foo.py
Hello
Hello
Hello
```

**Conditionals: If-Statements**

Format:
```python
if <boolean-expression>:
    <statement>...
else:
    <statement>...
```

Example:
```python
# Put x in z if it is positive
if x > y:
    z = x
else:
    z = y
```

Execution:
if `<boolean-expression>` is true, then execute all of the statements indented directly underneath (until first non-indented statement).

**Conditionals: If-Else-Statements**

Format:
```
if <boolean-expression>:
    <statement>...
else:
    <statement>...
```

Example:
```
# Put max of x, y in z
if x > y:
    z = x
else:
    z = y
```

Execution:
if `<boolean-expression>` is true, then execute statements indented under if; otherwise execute the statements indented under else.

**Conditionals: “Control Flow” Statements**

```
if b:
    s1 # statement
s3
```

Flow:
Program only takes one path each execution.
Program Flow vs. Local Variables

```python
def max(x, y):
    """Returns: max of x, y""
    if x > y:
        temp = x
        x = y
        y = temp
    return y
```

- temp is needed for swap
- x = y loses value of x
- "Scratch computation"
- Primary role of local vars
- max(3, 0):

```
x 0 y 3
temp 3
```

Program Flow and Testing

- Must understand which flow caused the error
- Unit test produces error
- Visualization tools show the current flow for error
- Visualization tools?
  - print statements
  - Advanced tools in IDEs (Integrated Dev. Environ)

```
# Put max of x, y in z
print 'before if'
if x > y:
    print 'if x>y'
    z = x
else:
    print 'else x<=y'
    z = y
print 'after if'
```

Watches vs. Traces

<table>
<thead>
<tr>
<th>Watch</th>
<th>Trace</th>
</tr>
</thead>
<tbody>
<tr>
<td>Visualization tool (e.g. print statement)</td>
<td>Visualization tool (e.g. print statement)</td>
</tr>
<tr>
<td>Looks at <strong>variable value</strong></td>
<td>Looks at <strong>program flow</strong></td>
</tr>
<tr>
<td>Often after an assignment</td>
<td>Before/after any point where flow can change</td>
</tr>
<tr>
<td>What you did in lab</td>
<td></td>
</tr>
</tbody>
</table>

Traces and Functions

```python
def shift(p):
    print 'Start shift()'
    p.x = p.y
    print p.x
    p.y = p.z
    print p.y
    p.z = p.x
    print p.z
    print 'End shift()'
```

**Example: flow.py**

```
# Put max of x, y, z in w
if x > y and x > z:
    w = x
elif y > z:
    w = y
else:
    w = z
```

Local Variables Revisited

- Never refer to a variable that might not exist
- Variable **scope**
  - Block (indented group) where it was first assigned
  - Way to think of variables: not actually part of Python
- Rule of Thumb: Limit variable usage to its scope

Conditionals: If-Elif-Else-Statements

**Format**

```
if <boolean-expression>:
    <statement>
    ...  
elif <boolean-expression>:
    <statement>
    ...
else:
    <statement>
```

**Example**

```
# Put max of x, y, z in w
if x > y and x > z:
    w = x
elif y > z:
    w = y
else:
    w = z
```