

Lecture 7

Conditionals & Control Flow

Announcements For This Lecture

Readings

- Sections 5.1-5.7 today
- Chapter 4 for Tuesday

Assignment 2

- Will cover function frames again on Thurs
 - My version \neq textbook
 - Posted “conversion” on today’s lecture page

Assignment 1

- Due Monday, Sep. 17th
 - Consultants all weekend!
 - One-on-ones ongoing
- Submit whatever you can
 - Will assess most by Tues
 - Will contact students with the most problems
 - But don’t take as a chance to procrastinate

Testing last_name_first(n)

```
# test procedure
```

```
def test_last_name_first():
```

```
    """Test procedure for last_name_first(n)"""
```

```
    unittest.assertEqual('White, Walker',  
                          last_name_first('Walker White'))
```

```
    unittest.assertEqual('White, Walker',  
                          last_name_first('Walker White'))
```

Expected is the
literal value.

Received is the
expression.

Quits Python
if not equal

```
# Application code
```

```
if __name__ == '__main__':
```

```
    test_last_name_first()
```

```
    print 'Module name is working correctly'
```

Message will print
out only if no errors.

Finding the Error

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

```
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 '+ `first`
```

```
    last = n[end_first+1:]
```

```
    print 'last is '+ `last`
```

```
    return last+', '+first
```

Print variable after
each assignment

Optional: Annotate
value to make it
easier to identify

Types of Testing

Black Box Testing

- Function is “opaque”
 - Test looks at what it does
 - **Fruitful**: what it returns
 - **Procedure**: what changes
- **Example**: Unit tests
- **Problems**:
 - Are the tests everything?
 - What caused the error?

White Box Testing

- Function is “transparent”
 - Tests/debugging takes place inside of function
 - Focuses on where error is
- **Example**: Use of print
- **Problems**:
 - Much harder to do
 - Must remove when done

Structure vs. Flow

Program Structure

- Way statements are presented
 - Order statements are listed
 - Inside/outside of a function
 - Will see other ways...
- Indicate possibilities over **multiple executions**

Program Flow

- Order statements are executed
 - Not the same as structure
 - Some statements duplicated
 - Some statements are skipped
- Indicates what really happens in a **single execution**

Have already seen this
difference with functions

Structure vs. Flow: Example

Program Structure

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

Statement
listed once

Program Flow

```
>>> python foo.py  
'Hello'  
'Hello'  
'Hello'
```

Statement
executed 3x

Bugs can occur when we
get a flow other than one
that we where expecting

Conditionals: If-Statements

Format

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

Example

```
# Put x in z if it is positive  
if x > 0:  
    z = x
```

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:

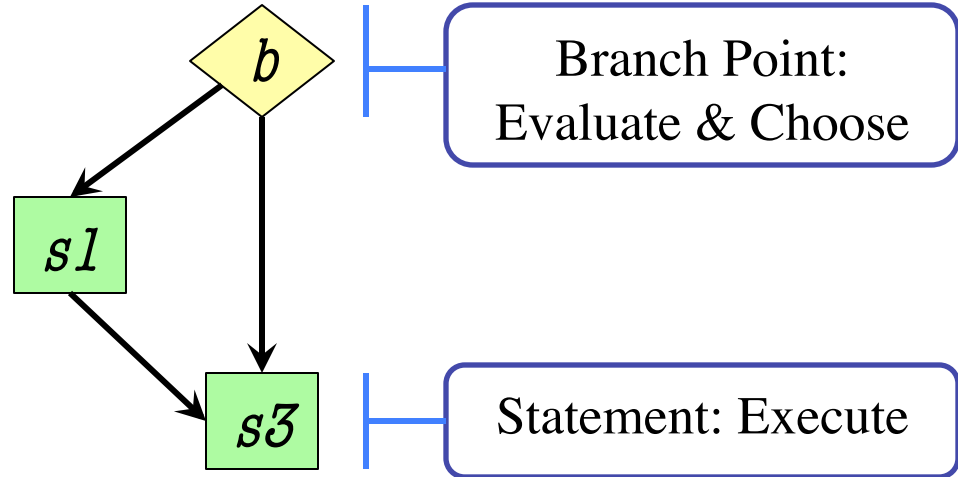
if <boolean-expression> is true, then execute statements indented under if; otherwise execute the statements indented under else

Conditionals: “Control Flow” Statements

if b :

| $s1$ # statement

| $s3$



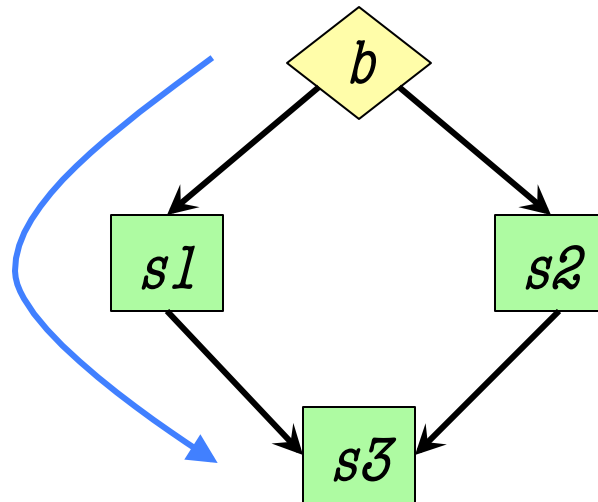
if b :

| $s1$

else:

| $s2$

| $s3$



Flow

Program only takes one path each execution

Program Flow vs. Local Variables

```
def max(x,y):
```

```
    """Returns: max of x, y"""
```

```
    # swap x, y
```

```
    # put the larger in y
```

```
    if x > y:
```

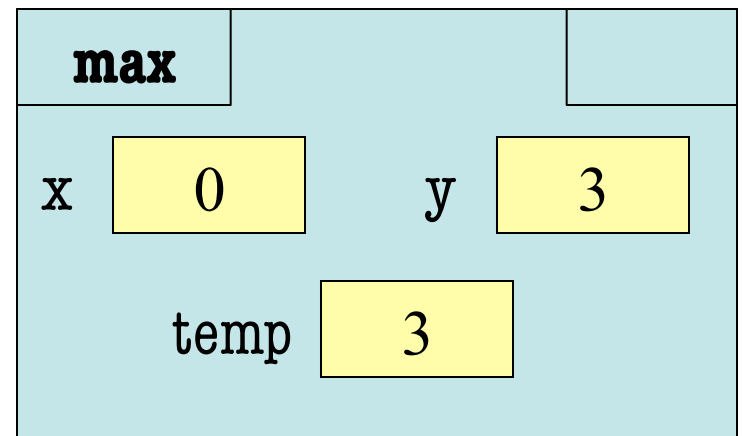
```
        temp = x
```

```
        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):



Program Flow vs. Local Variables

```
def max(x,y):
```

```
    """Returns: max of x, y"""
```

```
    # swap x, y
```

```
    # put the larger in y
```

```
    if x > y:
```

```
        temp = x
```

```
        x = y
```

```
        y = temp
```

```
    return temp
```

- Value of max(3,0)?

A: 3

B: 0

C: **Error!**

D: I do not know

Program Flow vs. Local Variables

```
def max(x,y):
```

```
    """Returns: max of x, y"""
```

```
    # swap x, y
```

```
    # put the larger in y
```

```
    if x > y:
```

```
        temp = x
```

```
        x = y
```

```
        y = temp
```

```
    return temp
```

- Value of max(3,0)?

A: 3 CORRECT

B: 0

C: Error!

D: I do not know

- Local variables last until
 - They are deleted or
 - End of the function
- Even if defined inside **if**

Program Flow vs. Local Variables

```
def max(x,y):
```

```
    """Returns: max of x, y"""
```

```
    # swap x, y
```

```
    # put the larger in y
```

```
    if x > y:
```

```
        temp = x
```

```
        x = y
```

```
        y = temp
```

```
    return temp
```

- Value of max(0,3)?

A: 3

B: 0

C: **Error!**

D: I do not know

Program Flow vs. Local Variables

```
def max(x,y):
```

```
    """Returns: max of x, y"""
```

```
    # swap x, y
```

```
    # put the larger in y
```

```
    if x > y:
```

```
        temp = x
```

```
        x = y
```

```
        y = temp
```

```
    return temp
```

- Value of max(0,3)?

A: 3

B: 0

C: **Error!** **CORRECT**

D: I do not know

- Variable existence depends on flow
- Understanding flow is important in testing

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'
```


Program Flow and Testing

- Call these tools **traces**
- No requirements on how to implement your traces
 - Less print statements ok
 - Do not need to word them exactly like we do
 - Do what ever is easiest for you to see the flow
- **Example:** flow.py

```
# 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

Watch

- Visualization tool (e.g. print statement)
- Looks at **variable value**
- Often after an assignment
- What you did in lab

Trace

- Visualization tool (e.g. print statement)
- Looks at **program flow**
- Before/after any point where flow can change

Traces and Watches

```
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

Watches

Traces

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

```
def max(x,y):
```

```
    """Returns: max of x, y"""
```

```
    # swap x, y
```

```
    # put larger in temp
```

```
    if x > y:
```

```
        temp = x
```

```
        x = y
```

```
        y = temp
```

```
    return temp
```

First assigned

Outside scope

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

```
def max(x,y):
```

```
    """Returns: max of x, y"""
```

```
    # swap x, y
```

```
    # put larger in temp
```

```
    temp = y
```

```
    if x > y:
```

```
        | temp = x
```

```
    return temp
```

First assigned

Inside scope

Variation on max

```
def max(x,y):
```

```
    """Returns:  
    max of x, y"""
```

```
    if x > y:
```

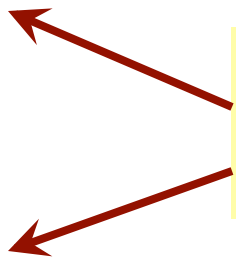
```
        return x
```

```
    else:
```

```
        return y
```

Which is better?
Matter of preference

There are two **returns!**
But only one is executed



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
```

Conditionals: If-Elif-Else-Statements

Format

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

Notes on Use

- No limit on number of elif
 - Can have as many as want
 - Must be between if, else
- The else is always optional
 - if-elif by itself is fine
- Booleans checked in order
 - Once it finds one that is true, it skips over all the others
 - The else means **all** are false

Conditional Expressions

Format

`e1 if bexp else e2`

- `e1` and `e2` are any expression
- `bexp` is a boolean expression
- This is an expression!

Example

`# Put max of x, y in z`

`z = x if x > y else y`



expression,
not statement