

Lecture 7

Conditionals & Control Flow

Announcements For This Lecture

Assignment 1

- Should be working on it
 - Have covered everything
 - Look at **lab** for more help
- Due Sunday at mid.
 - No new lab Thu/Fri
 - Can work at it during lab
- One-on-Ones ongoing
 - Lots of spaces available

Partners

- **You** must pair in CMS
- Go into the submission
 - Request your partner
 - Other person accepts

AI Quiz

- Sent out several e-mails
- Will drop next week!

Testing last_name_first(n)

test procedure

```
def test_last_name_first():
```

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

```
    result = name.last_name_first('Walker White')
```

```
    introcs.assert_equals('White, Walker', result)
```

```
    result = name.last_name_first('Walker White')
```

```
    introcs.assert_equals('White, Walker', result)
```

Call function
on test input

Compare to
expected output

Script code

```
test_last_name_first()
```

```
print('Module name passed all tests.')
```

Call test procedure
to activate the test

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

Types of Testing

Black Box Testing

- Function is “opaque”
 - Test looks at what it does
 - Works on functions you did not define
 - Tests for errors you did not anticipate
- **Example:** Testing a function that calculates the area of a circle
- **Problems:**
 - Are the tests everything?
 - What caused the error?

White Box Testing

- Function is “transparent”
 - Test looks at how it does it
 - Can find the bug location in function
 - Tests for errors you did not anticipate
- **Example:** Testing a function that calculates the area of a circle
- **Problems:**
 - Much harder to do
 - Must remove when done

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-name, first-name' """
```

```
    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
```

Print variable after
each assignment

Optional: Annotate
value to make it
easier to identify

How to Use the Results

- Goal of **white box testing** is **error location**
 - Want to identify the **exact line** with the error
 - Then you look ‘real hard’ at the line to find error
 - What you are doing in lab next week
- But similar approach to **black box testing**
 - At each line you have **expected** print result
 - Compare it to the **received** print result
 - Line before first mistake is *likely* the error

Warning About Print Statements

- Must remove them when you are done
 - Not part of the specification (violation)
 - Slow everything down unnecessarily
 - **App Store** will reject an app with prints
- But you might want them again later
 - **Solution**: “comment them out”
 - You can always uncomment later

Structure vs. Flow

Program Structure

- Order code is **presented**
 - Order statements are listed
 - Inside/outside of function
 - Will see other ways...
- Defines possibilities over **multiple executions**

Program Flow

- Order code is **executed**
 - Not the same as structure
 - Some statements duplicated
 - Some statements skipped
- Defines what happens in a **single execution**

Have already seen this
difference with functions

Structure vs. Flow: Example

Program Structure

```
def foo():
```

```
    print('Hello')
```

Statement
listed once

```
# Script Code
```

```
foo()
```

```
foo()
```

```
foo()
```

Program Flow

```
> python foo.py
```

```
'Hello'
```

```
'Hello'
```

```
'Hello'
```

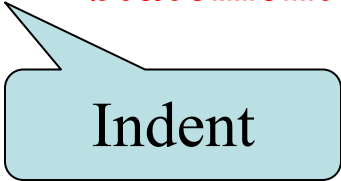
Statement
executed 3x

Bugs occur when flow does
not **match** expectations

Conditionals: If-Statements

Format

```
if expression:  
    statement  
    ...  
    statement
```



Indent

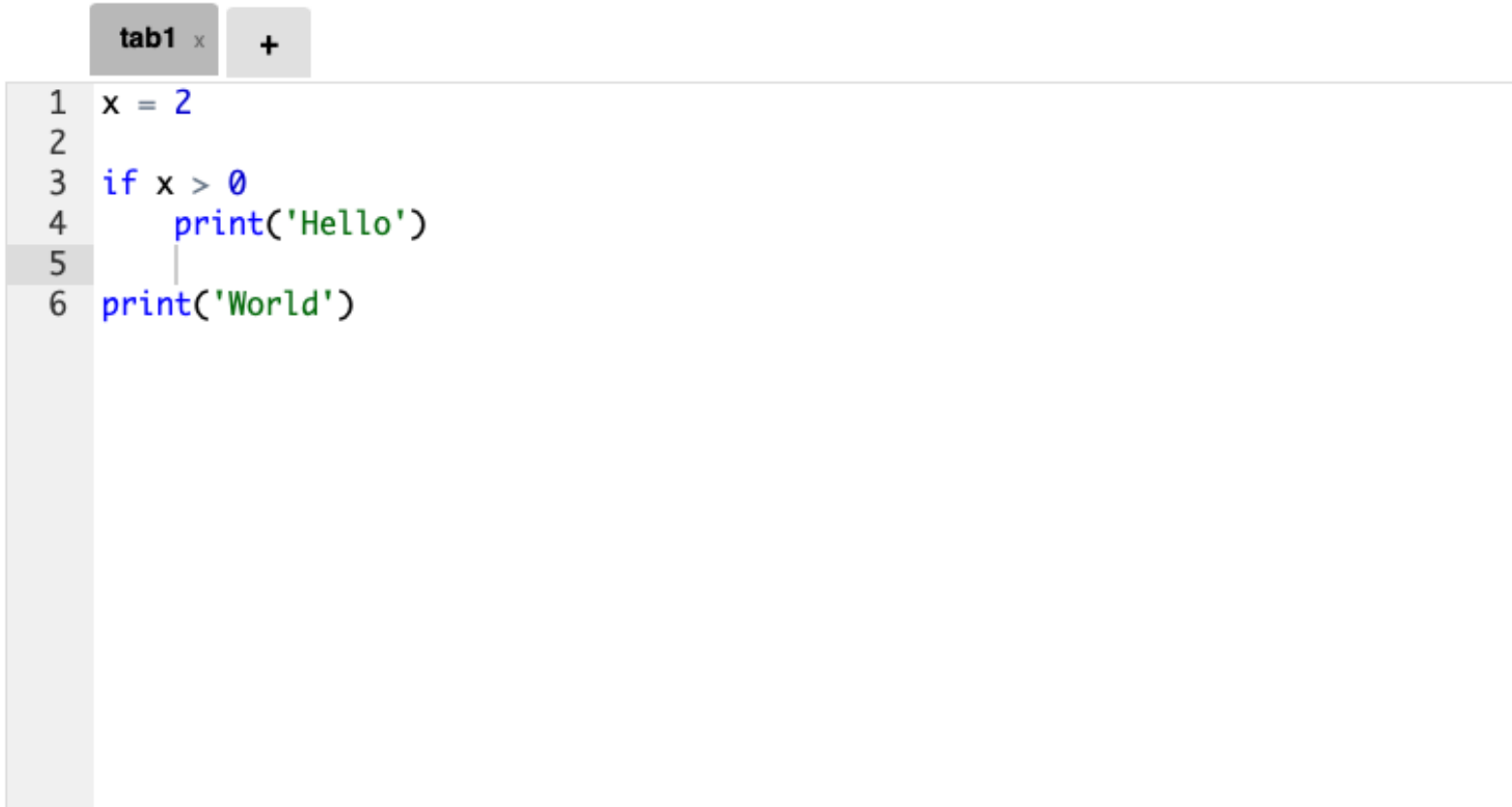
Example

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

Execution:

If *expression* is **True**, execute all statements **indented** underneath

Python Tutor Example



```
1 x = 2
2
3 if x > 0
4     print('Hello')
5
6 print('World')
```

Double click the tab to change name, press enter when done.

Visualize

Execute Code

Edit Code

Conditionals: If-Else-Statements

Format

```
if expression :  
    statement  
    ...  
else:  
    statement  
    ...
```

Example

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

Execution:

If *expression* is **True**, execute all statements indented under **if**.
If *expression* is **False**, execute all statements indented under **else**.

Python Tutor Example



```
1 x = 2
2
3 if x > 0
4     print('Hello')
5 else:
6     print('Good-bye')
7
8 print('World')
```

Double click the tab to change name, press enter when done.

Visualize

Execute Code

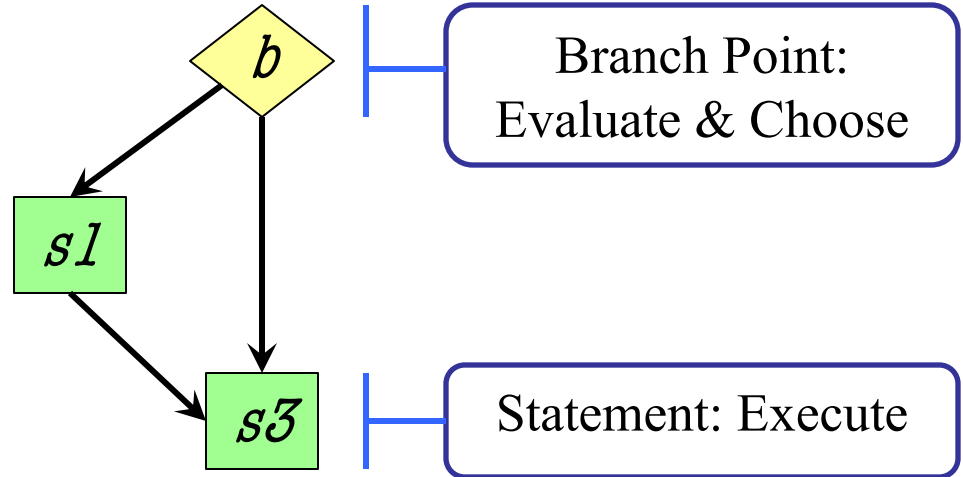
Edit Code

Conditionals: “Control Flow” Statements

if b :

| $s1$ # statement

$s3$



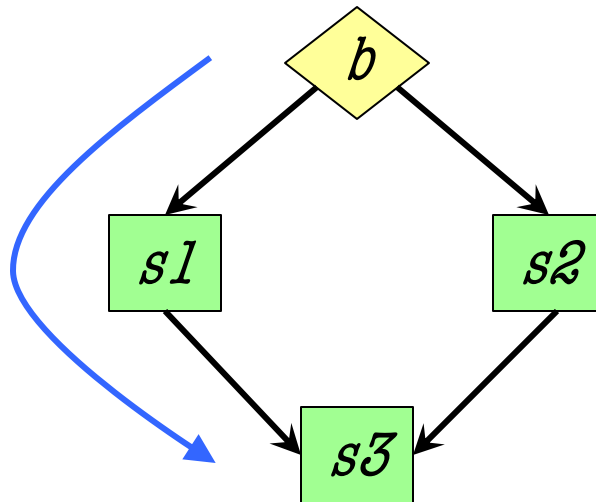
if b :

| $s1$

else:

| $s2$

$s3$



Flow

Program only takes one path each execution

Program Flow and Call Frames

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

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

```
    # simple implementation
```

```
1  if x > y:
```

```
2  |     return x
```

```
3  return y
```

```
max(0,3):
```

max		1
x	0	
y	3	

Frame sequence
depends on flow

Program Flow and Call Frames

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

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

```
    # simple implementation
```

```
1  if x > y:
```

```
2  |     return x
```

```
3  return y
```

Frame sequence
depends on flow

```
max(0,3):
```

max		3
x	0	
y	3	

Skips line 2

Program Flow and Call Frames

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

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

```
    # simple implementation
```

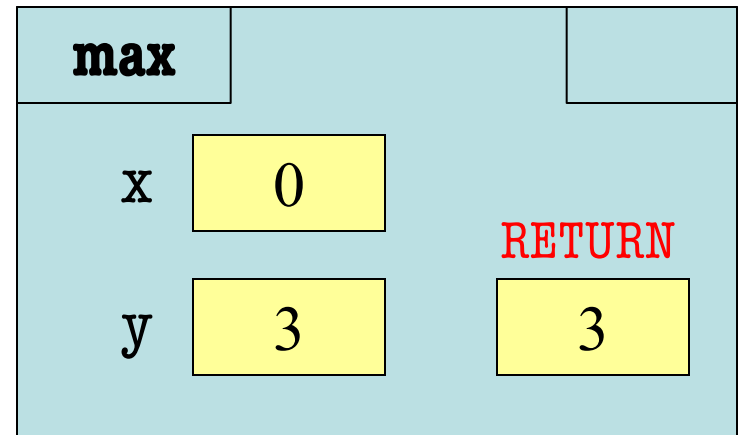
```
1  if x > y:
```

```
2  |     return x
```

```
3  return y
```

Frame sequence
depends on flow

```
max(0,3):
```



Skips line 2

Program Flow vs. Local Variables

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

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

```
    # swap x, y
```

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

```
1  if x > y:
```

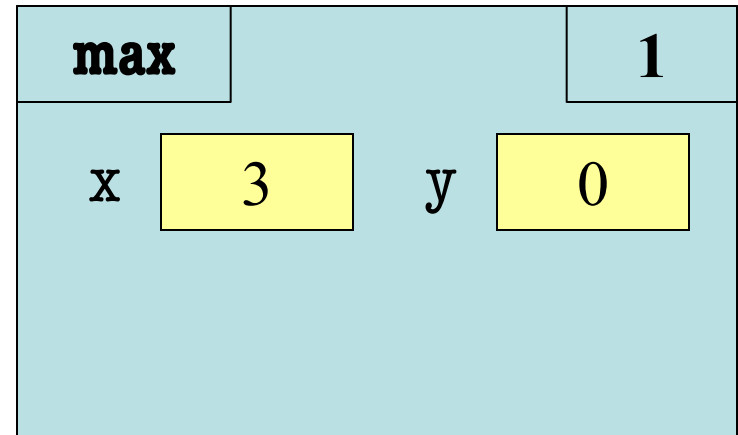
```
2      temp = x
```

```
3      x = y
```

```
4      y = temp
```

```
5  return y
```

- max(3,0):



Swaps max
into var y

Program Flow vs. Local Variables

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

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

```
    # swap x, y
```

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

```
1  if x > y:
```

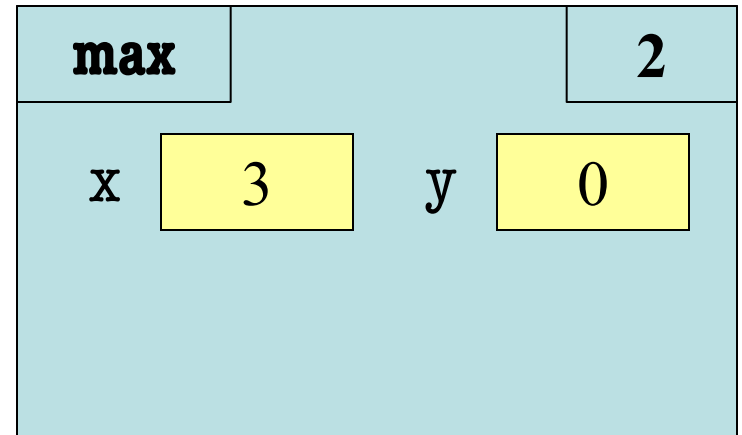
```
2      temp = x
```

```
3      x = y
```

```
4      y = temp
```

```
5  return y
```

- max(3,0):



Swaps max
into var y

Program Flow vs. Local Variables

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

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

```
    # swap x, y
```

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

```
1  if x > y:
```

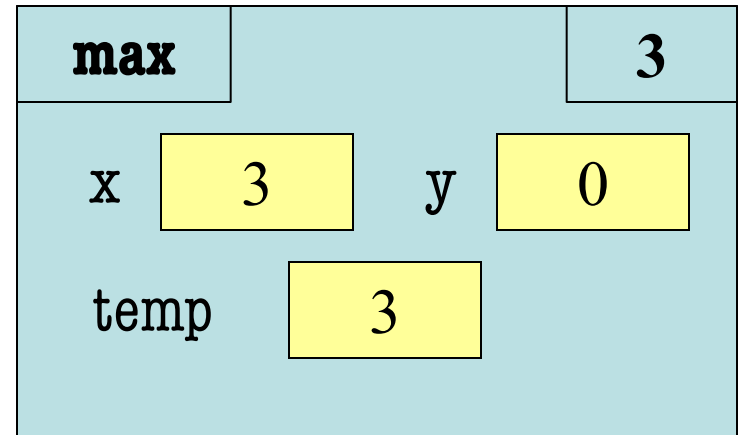
```
2      temp = x
```

```
3      x = y
```

```
4      y = temp
```

```
5  return y
```

- max(3,0):



Swaps max
into var y

Program Flow vs. Local Variables

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

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

```
    # swap x, y
```

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

```
1  if x > y:
```

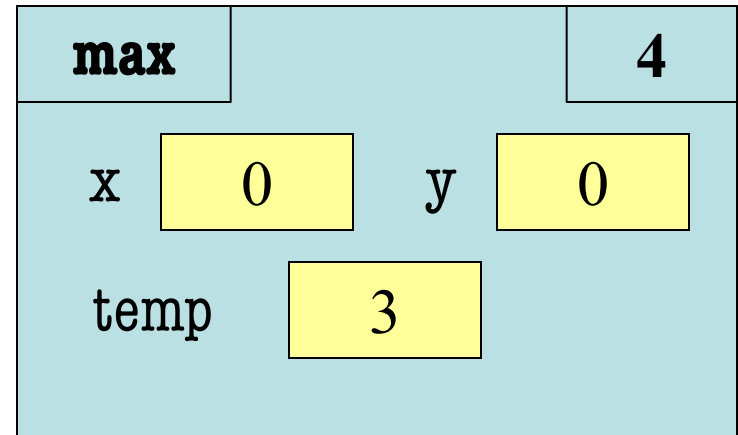
```
2      temp = x
```

```
3      x = y
```

```
4      y = temp
```

```
5  return y
```

- max(3,0):



Swaps max
into var y

Program Flow vs. Local Variables

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

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

```
    # swap x, y
```

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

```
1  if x > y:
```

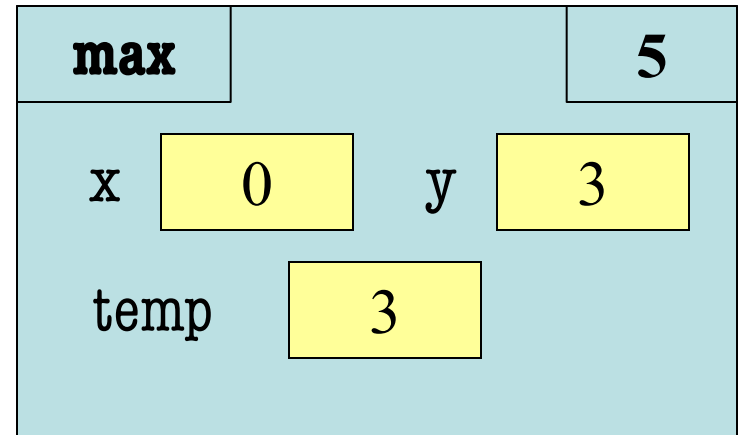
```
2      temp = x
```

```
3      x = y
```

```
4      y = temp
```

```
5  return y
```

- max(3,0):



Swaps max
into var y

Program Flow vs. Local Variables

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

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

```
    # swap x, y
```

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

```
1  if x > y:
```

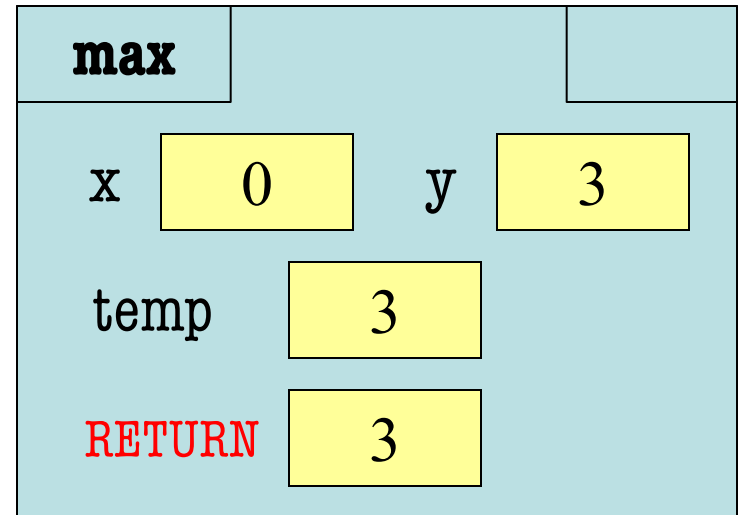
```
2      temp = x
```

```
3      x = y
```

```
4      y = temp
```

```
5  return y
```

- max(3,0):



Swaps max
into var y

Program Flow vs. Local Variables

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

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

```
    # swap x, y
```

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

```
1  if x > y:
```

```
2      temp = x
```

```
3      x = y
```

```
4      y = temp
```

```
5  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
```

```
1  if x > y:
```

```
2      temp = x
```

```
3      x = y
```

```
4      y = temp
```

```
5  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
```

```
1  if x > y:
```

```
2      temp = x
```

```
3      x = y
```

```
4      y = temp
```

```
5  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
```

```
1  if x > y:
```

```
2      temp = x
```

```
3      x = y
```

```
4      y = temp
```

```
5  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

Testing and Code Coverage

- Typically, tests are written from **specification**
 - This is because they should be written first
 - You run these tests while you implement
- But sometimes tests leverage code structure
 - You know the control-flow branches
 - You want to make sure each branch is correct
 - So you explicitly have a test for **each branch**
- This is called **code coverage**

Which Way is Correct?

- Code coverage requires knowing code
 - So it must be done after implementation
 - But best practice is to write tests *first*
- Do them **BOTH**
 - Write tests from the specification
 - Implement the function while testing
 - Go back and add tests for full coverage
 - Ideally this does not require adding tests

Recall: Debugging

- 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-name, first-name' """
```

```
    end_first = n.find(' ')
```

```
    print(end_first)
```

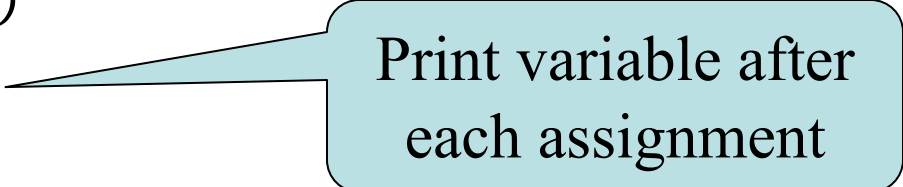
```
    first = n[:end_first]
```

```
    print('first is '+str(first))
```

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

```
    print('last is '+str(last))
```

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



Print variable after
each assignment



Called watches

Now Have a Different Challenge

```
# 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')
```

- What was executed?
 - The **if** -statement?
 - Or the **else**-statement?
- More print statements
 - **Trace** program flow
 - Verify flow is correct

Called traces

Watches vs. Traces

Watch

- Visualization tool
 - Often print/log statement
 - May have IDE support
- Looks at **variable value**
 - Anywhere it can change
 - Often after assignment

Trace

- Visualization tool
 - Often print/log statement
 - May have IDE support
- Looks at **program flow**
 - Anywhere it can change
 - Before/after control

Traces and Watches

```
print('before if')
```

Example: flow.py

```
if x > y:
```

```
    print('if x>y')
```

```
    z = y
```

```
    print(z)
```

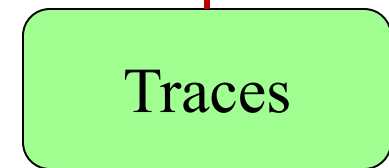
```
else:
```

```
    print('else x<=y')
```

```
    z = y
```

```
    print(z)
```

```
print('after if')
```



Conditionals: If-Elif-Else-Statements

Format

```
if expression :  
    statement  
    ...  
elif 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 expression :  
    statement  
    ...  
elif 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 first True, skips over all others
 - **else** means **all** are false

Python Tutor Example



The image shows a Python Tutor interface. At the top, there is a tab labeled 'tab1' with a close button 'x' and a plus sign '+'. Below the tab is a code editor with the following Python code:

```
1 x = 2
2
3 if x > 0
4     print('Hello')
5 elif x < 0:
6     print('Whatever')
7 else:
8     print('Good-bye')
9
10 print('World')
```

The line `print('Whatever')` on line 6 is highlighted with a light blue background.

Double click the tab to change name, press enter when done.

Visualize

Execute Code

Edit Code

Conditional Expressions

Format

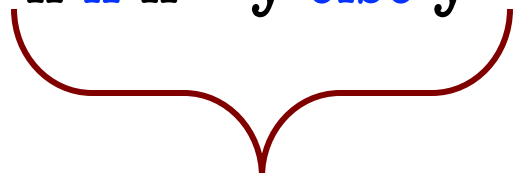
e1 **if** bexp **else** e2

- e1 and e2 are *any* expression
- bexp is a boolean expression
- This is an expression!
 - **Evaluates** to e1 if bexp True
 - **Evaluates** to e2 if bexp False

Example

Put max of x, y in z

z = x **if** x > y **else** y



expression,
not statement