Module 9

Conditionals

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')

# Script Code
foo()
foo()
foo()
```

statement listed once

Program Flow

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

statement executed 3x

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

Why Is This Important

- You have been writing “straight-line” code
  - Every line of code you write executed in order
  - Functions mainly used to group code together
- But it is possible to control program flow
  - Ask Python to skip over statements
  - Ask Python to repeat statements
- This requires a **control-flow statement**
  - Category of statements; not a single type
  - This video series will cover the **conditional**

Conditionals: If-Statements

Format

```
if expression :
    statement
    …
    statement
```

Indent

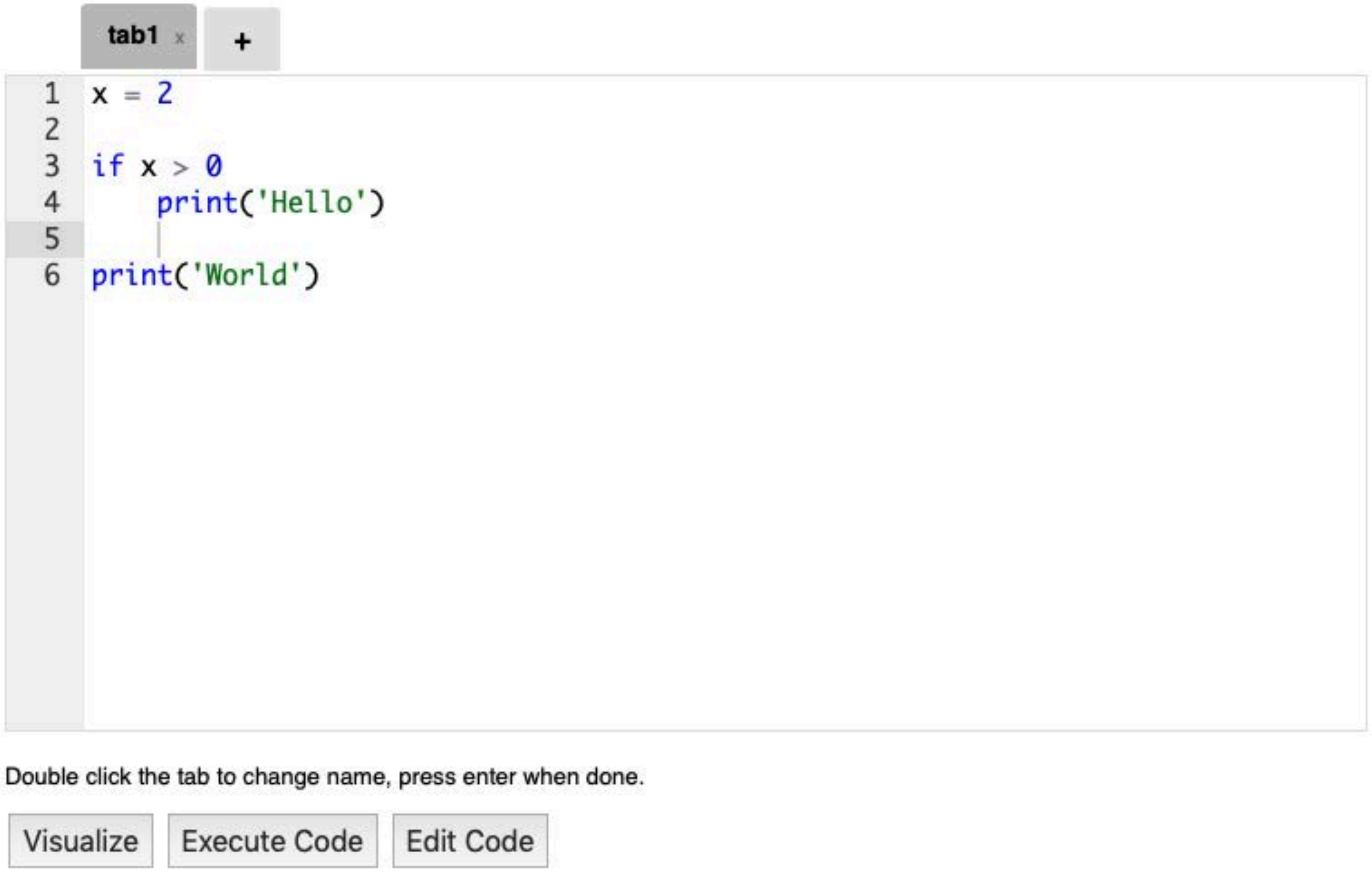
Example

```
# 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

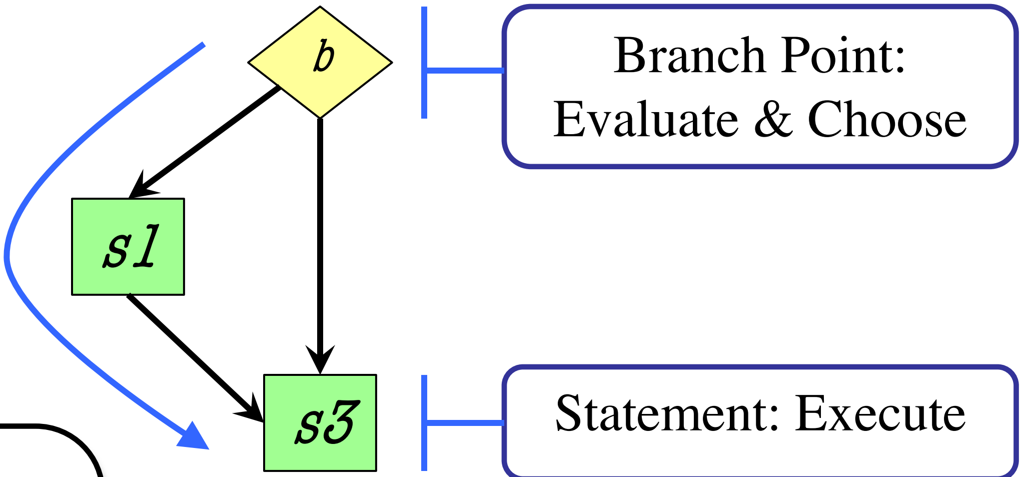


```
x = 2

if x > 0
    print('Hello')

print('World')
```

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

Conditionals: “Control Flow” Statements

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



Flow

Program only takes one path each execution

Conditionals: If-Else-Statements

Format

```
if expression :
        statement
        …
else:
        statement
        …
```

Example

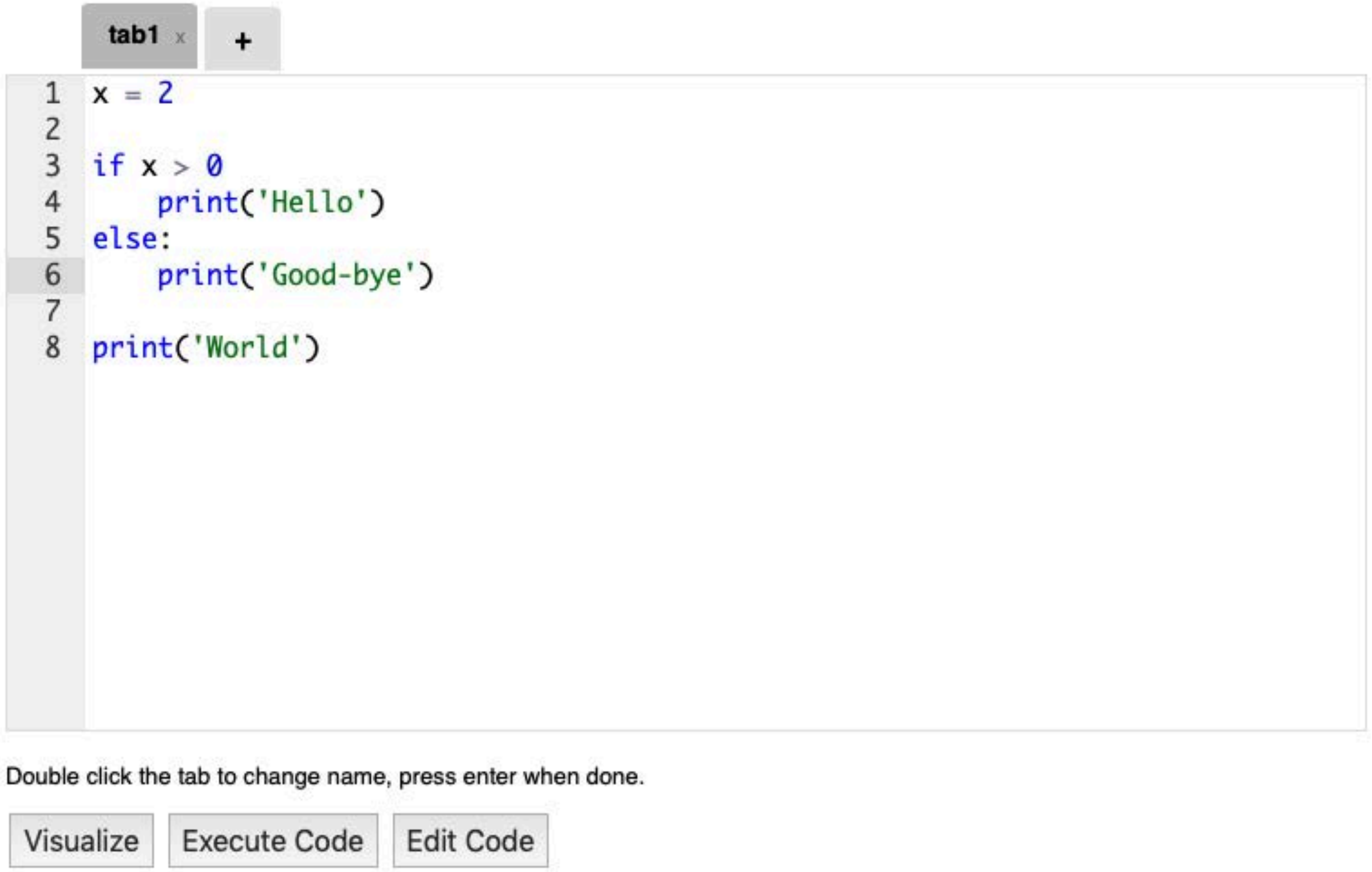
```
# Put max of x, y in z
if x > y:
    z  = x
else:
    z  = y
```

Execution:

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

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

Python Tutor Example

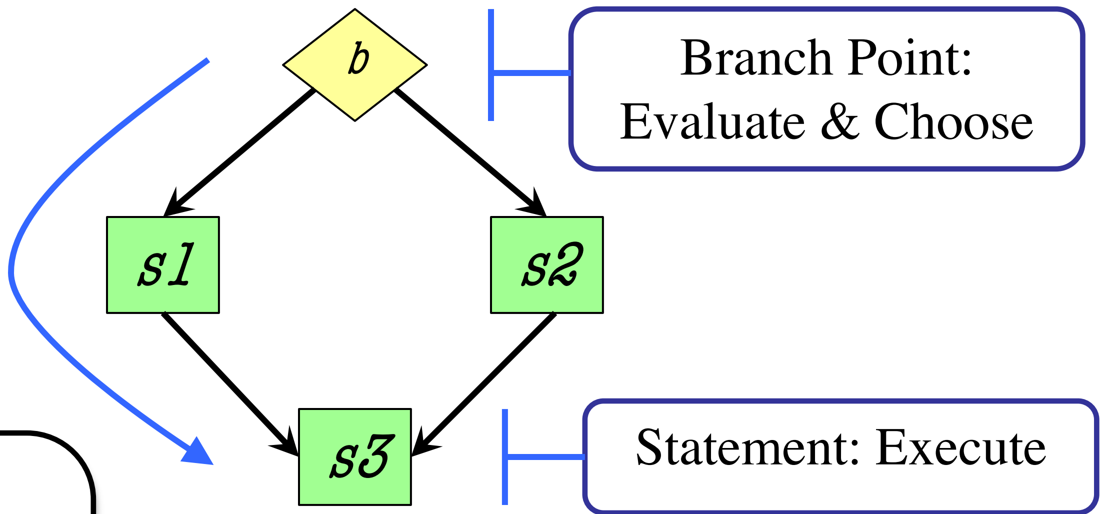


```
x = 2

if x > 0
    print('Hello')
else:
    print('Good-bye')

print('World')
```

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

Conditionals: "Control Flow" Statements

```
if b:
    s1
else:
    s2
s3
```



Flow

Program only takes one path each execution

Conditionals: If-Elif-Else-Statements

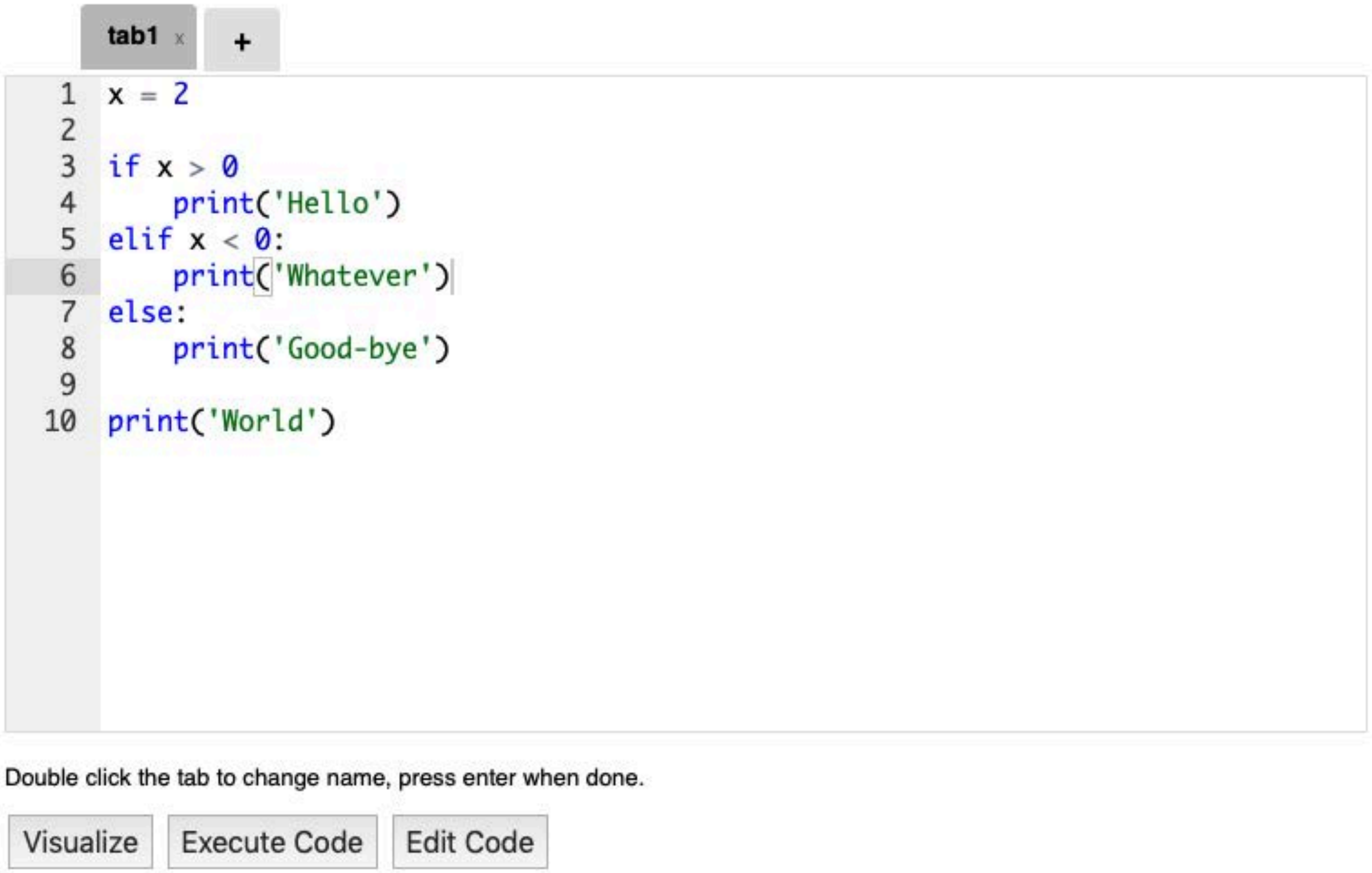
Format

```
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
```

Python Tutor Example



```
x = 2

if x > 0
    print('Hello')
elif x < 0:
    print('Whatever')
else:
    print('Good-bye')

print('World')
```

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

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

Problem Statement

- Common pattern: if-statements w/ assignments
  - Need to assign a value to a single variable
  - But the actual value depends on the flow
- **Example**:

```
if x > y:
    z = x
else:
    z = y
```

All to assign to z.
Is there an easier way?

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
```

$$\texttt{z = }\underbrace{\texttt{x if x > y else y}}_{\text{expression, not statement}}$$

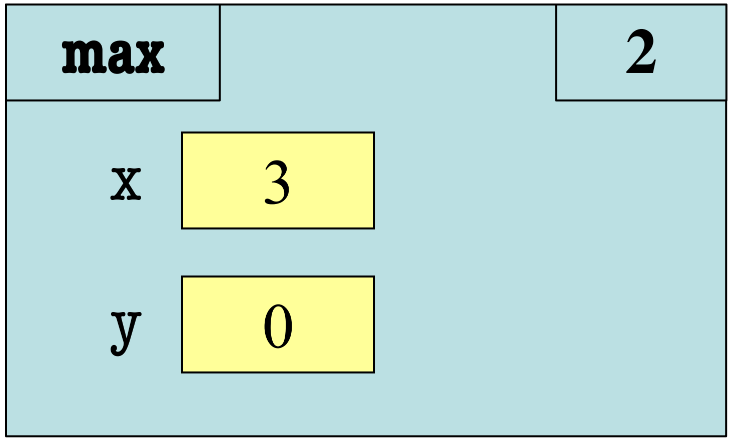
Using Conditionals

- Conditionals: when variables are unknown
  - Conditionals test different possibilities
  - If you always know value, only one choice
- When can variables be unknown?
  - When they are the result of user input
  - When they are the result of a function call
- Conditionals are a natural fit for functions

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(3,0):



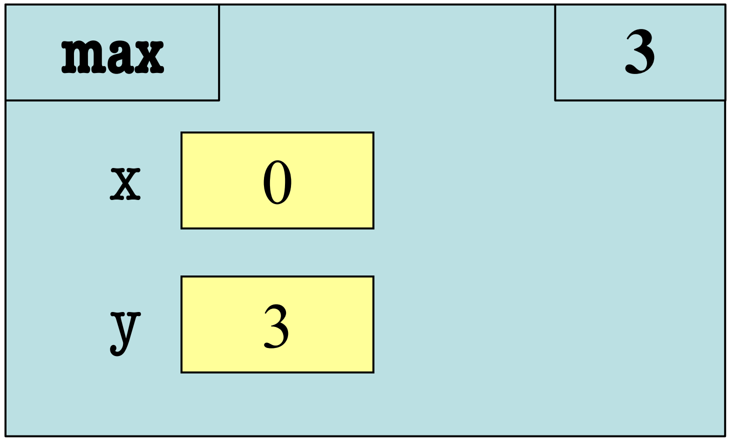
Frame sequence depends on flow

Reaches 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
```

max(0,3):



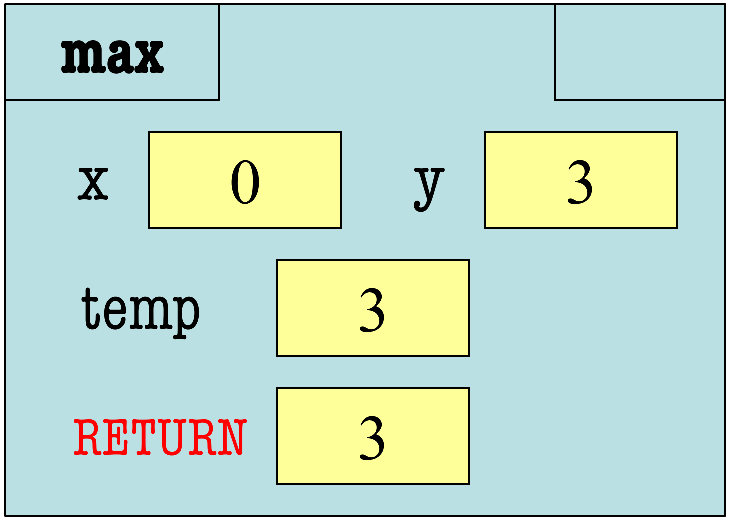
Frame sequence depends on flow

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
    if x > y:
        temp = x
        x = y
        y = temp

    return y
```

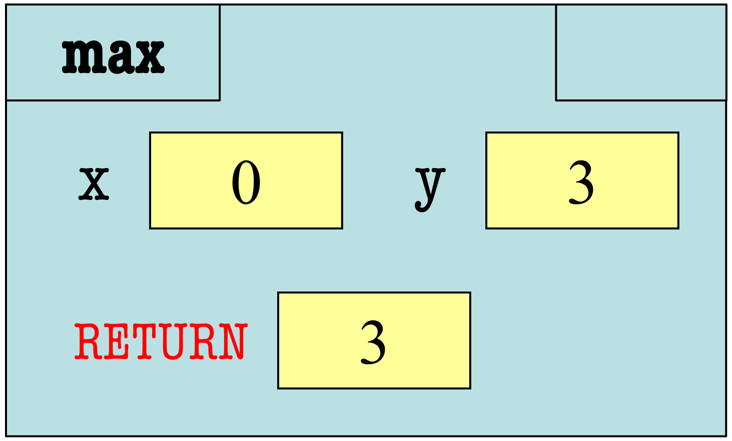
- temp is needed for swap
  - x = y loses value of x
  - "Scratch computation"
  - Primary role of local vars
- max(3,0):



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 y
```

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



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

- 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(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

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

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

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**

A Simple Example

```
def anglicize(n):
    """Returns: English equiv of n

    Precondition: n in 1..19"""
    if n == 1:
        return 'one'
    …
    elif n == 18:
        return 'eighteen'

    return 'nineteen'
```

Need a test for each "branch"

Walkthrough test script for anglicize()

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: 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(str(first))
    last  = n[end_first+1:]
    print(str(last))
    return last+', '+first
```

Print variable after each assignment

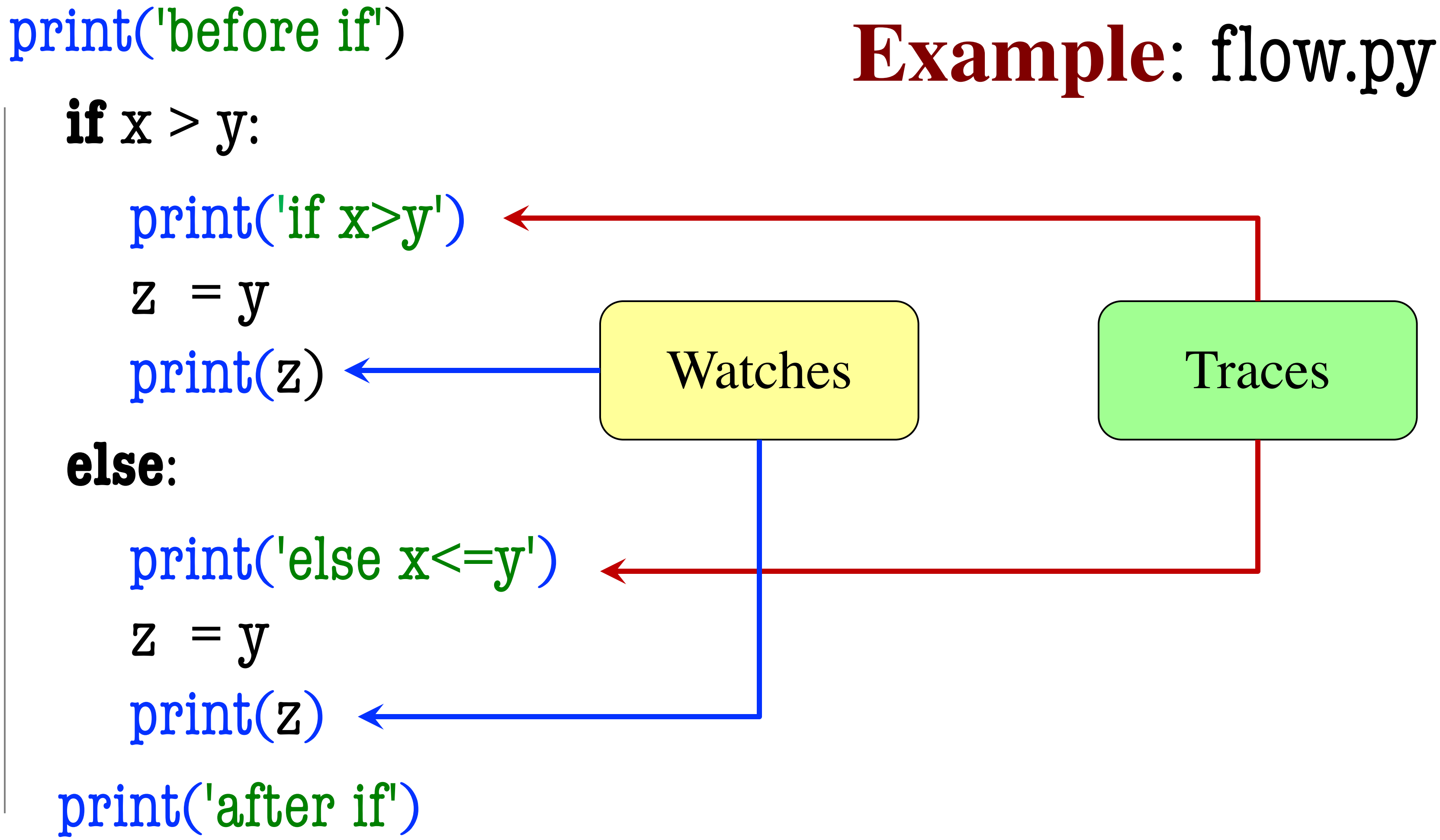
Necessary because do not always have Tutor

Visualizing Code

- These print statements are called **Watches**
  - Looks at **variable value** after assignment
  - It is watching for any possible changes
- But now we have a different problem
  - Program flow can take many paths
  - Often unsure of which path taken
  - Want print statements to trace code path
- Obviously these are called **Traces**

Traces and Functions

Example: flow.py



```
print('before if')
if x > y:
    print('if x>y')
    z  = y
    print(z)
else:
    print('else x<=y')
    z  = y
    print(z)
print('after if')
```

Scripts vs. Modules

- The difference is how to use the file
  - Modules are meant to be imported
  - Scripts are run from command line
- But sometimes want to import a script
  - Want access to functions in the script
  - But do not want to run the whole script
- **Example:** Test scripts
  - Each test is its own procedure

Idea: Conditional Execution

- Want script to NOT execute on import
  - **Script Code:** code at the bottom of file
  - Typically calls functions defined
- Can do this with an if-statement

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

Special pre-assigned variable when run

Name assigned when run as script

- Demo with test script