

Lecture 3

Defining Functions

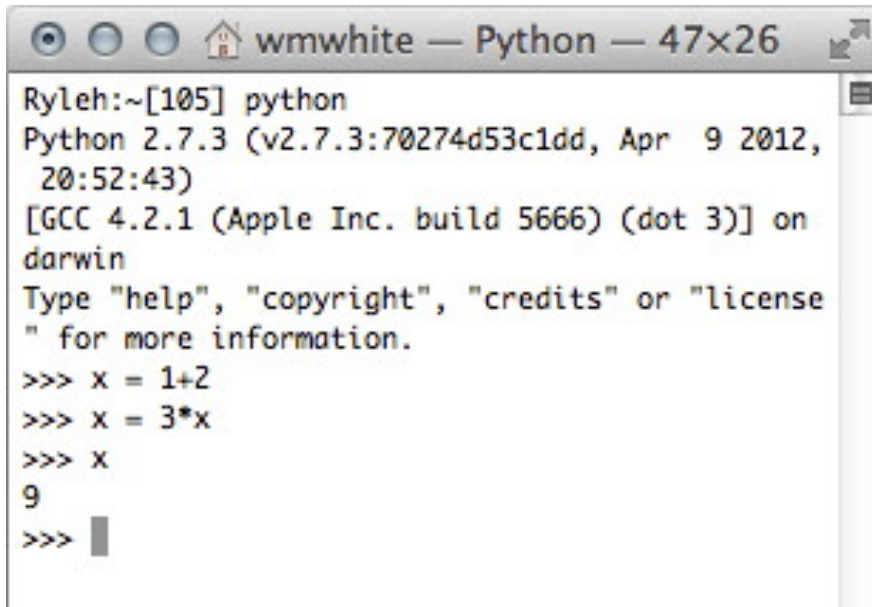
Recall: Modules

- Modules provide extra functions, variables
 - **Example:** math provides `math.cos()`, `math.pi`
 - Access them with the `import` command
- Python provides a lot of them for us
- **This Lecture:** How to make modules
 - Komodo Edit to *make* a module
 - Python to *use* the module



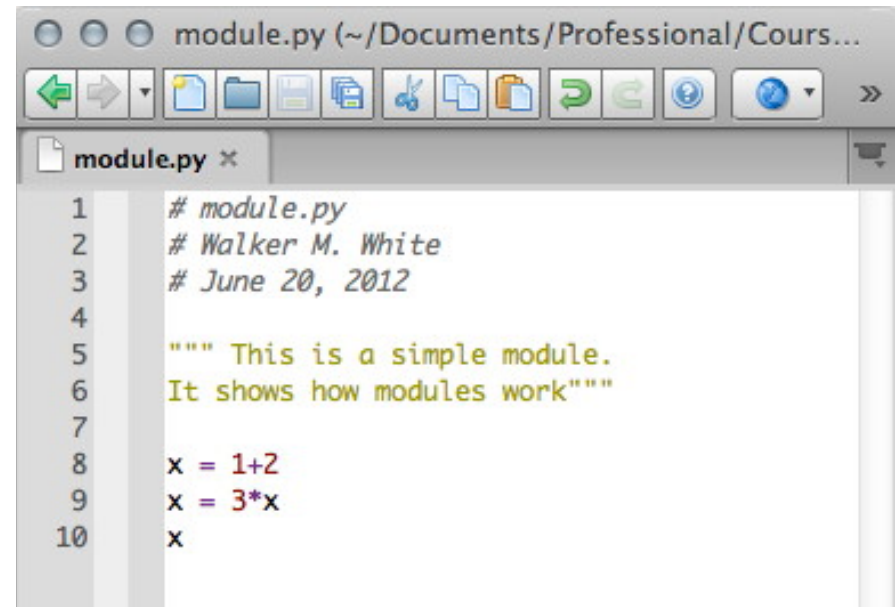
Two different programs

Python Shell vs. Modules



```
Ryleh:~[105] python
Python 2.7.3 (v2.7.3:70274d53c1dd, Apr 9 2012,
20:52:43)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on
darwin
Type "help", "copyright", "credits" or "license"
for more information.
>>> x = 1+2
>>> x = 3*x
>>> x
9
>>>
```

- Launch in command line
- Type each line separately
- Python executes as you type



```
1 # module.py
2 # Walker M. White
3 # June 20, 2012
4
5 """ This is a simple module.
6 It shows how modules work"""
7
8 x = 1+2
9 x = 3*x
10 x
```

- **Write in a text editor**
 - We use Komodo Edit
 - But anything will work
- Run module with import

Using a Module

Module Contents

```
# module.py
```

Single line comment
(not executed)

```
""" This is a simple module.  
It shows how modules work"""
```

Docstring (note the Triple Quotes)
Acts as a multiple-line comment
Useful for *code documentation*

```
x = 1+2
```

```
x = 3*x
```

Commands
Executed on import

```
x
```

Not a command.
import **ignores this**

Using a Module

Module Contents

```
# module.py
```

```
""" This is a simple module.  
It shows how modules work"""
```

```
x = 1+2
```

```
x = 3*x
```

```
x
```

“**Module data**” must be
prefixed by module name

Prints **docstring** and
module contents

Python Shell

```
>>> import module
```

```
>>> x
```

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

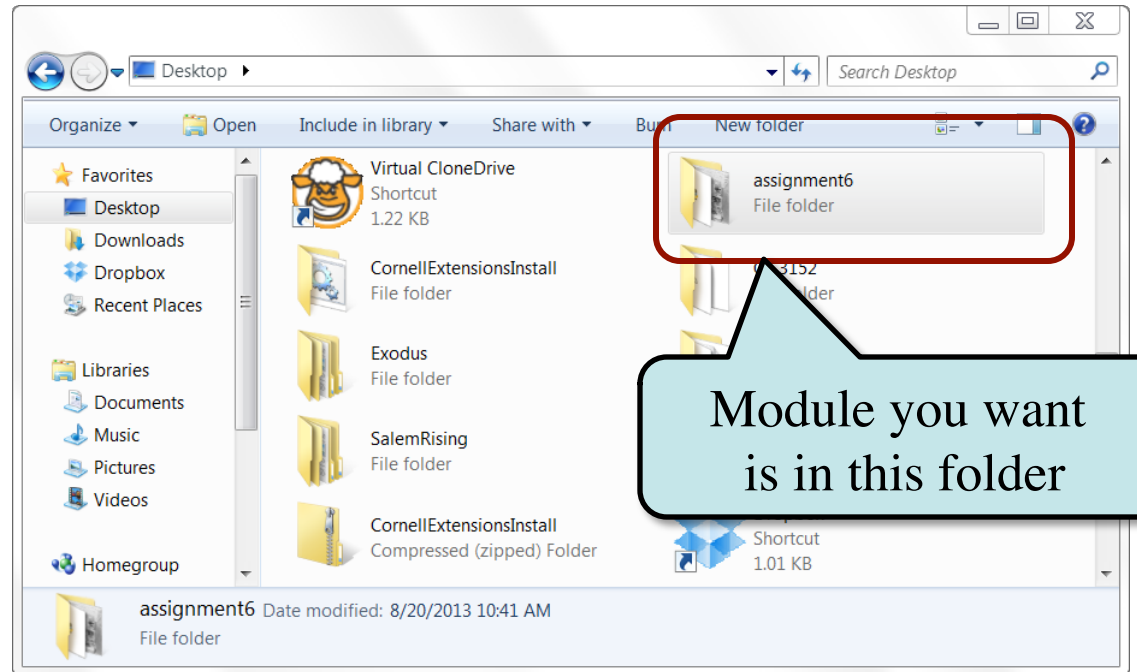
NameError: name 'x' is not defined

```
>>> module.x
```

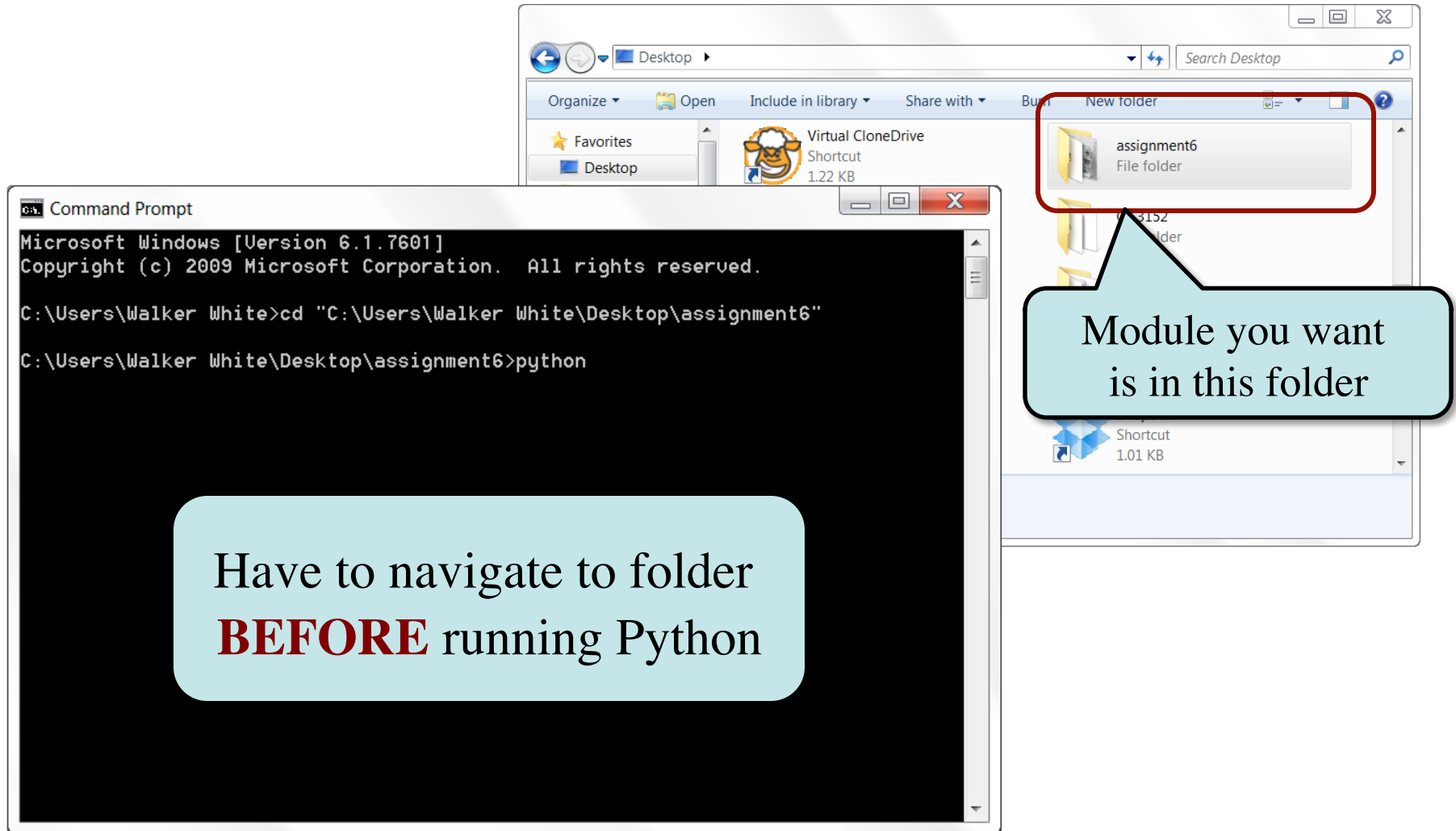
```
9
```

```
>>> help(module)
```

Modules Must be in Working Directory!



Modules Must be in Working Directory!



We Write Programs to Do Things

- Functions are the **key doers**

Function Call

- Command to **do** the function

`greet('Walker')`

argument to
assign to n

Function
Header

Function Definition

- Defines what function **does**

`def greet(n):`

`print 'Hello '+n+'!'`

declaration of
parameter n

Function
Body
(indented)

- **Parameter:** variable that is listed within the parentheses of a method header.
- **Argument:** a value to assign to the method parameter when it is called

Anatomy of a Function Definition

name

parameters

def greet(n): Function Header

"""Prints a greeting to the name n

Precondition: n is a string
representing a person's name"""

Docstring
Specification

print 'Hello '+n+'!'

print 'How are you?'

Statements to
execute when called

The vertical line
indicates indentation

Use vertical lines when you write Python
on **exams** so we can see indentation

Procedures vs. Fruitful Functions

Procedures

- Functions that **do** something
- Call them as a **statement**
- Example: `greet('Walker')`

Fruitful Functions

- Functions that give a **value**
- Call them in an **expression**
- Example: `x = round(2.56,1)`

Historical Aside

- Historically “function” = “fruitful function”
- But now we use “function” to refer to both

The **return** Statement

- Fruitful functions require a **return statement**
- **Format:** `return <expression>`
 - Provides value when call is used in an expression
 - Also stops executing the function!
 - Any statements after a **return** are ignored
- **Example:** temperature converter function

```
def to_centrigrade(x):
```

```
    """Returns: x converted to centigrade"""
```

```
    return 5*(x-32)/9.0
```

Functions and Modules

- Purpose of modules is **function definitions**
 - Function definitions are written in module file
 - Import the module to call the functions
- Your Python workflow (right now) is

1. Write a function in a module (a .py file)
2. Open up the command shell
3. Move to the directory with this file
4. Start Python (type python)
5. Import the module
6. Try out the function

Aside: Constants

- Modules often have variables outside a function
 - We call these global variables
 - Accessible once you import the module
- Global variables should be **constants**
 - Variables that never, ever change
 - Mnemonic representation of important value
 - **Example:** `math.pi`, `math.e` in `math`
- In this class, constant names are **capitalized!**
 - So we can tell them apart from non-constants

Module Example: Temperature Converter

```
# temperature.py
```

```
"""Conversion functions between fahrenheit and centigrade"""
```

```
# Functions
```

```
def to_centigrade(x):
```

```
    """Returns: x converted to centigrade"""
```

```
    return 5*(x-32)/9.0
```

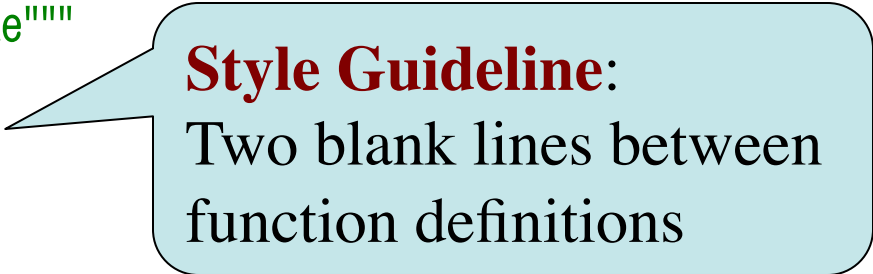
```
def to_fahrenheit(x):
```

```
    """Returns: x converted to fahrenheit"""
```

```
    return 9*x/5.0+32
```

```
# Constants
```

```
FREEZING_C = 0.0  # temp. water freezes
```



Style Guideline:
Two blank lines between
function definitions

String Parsing Example

def second_in_list(s):

"""Returns: second item in comma-separated list

The final result does not have any whitespace on edges

Precondition: s is a string of items separated by a comma."""

startcomma = s.index(',')

tail = s[startcomma+1:]

endcomma = tail.index(',')

item = tail[:endcomma].strip()

return item

See commalist.py

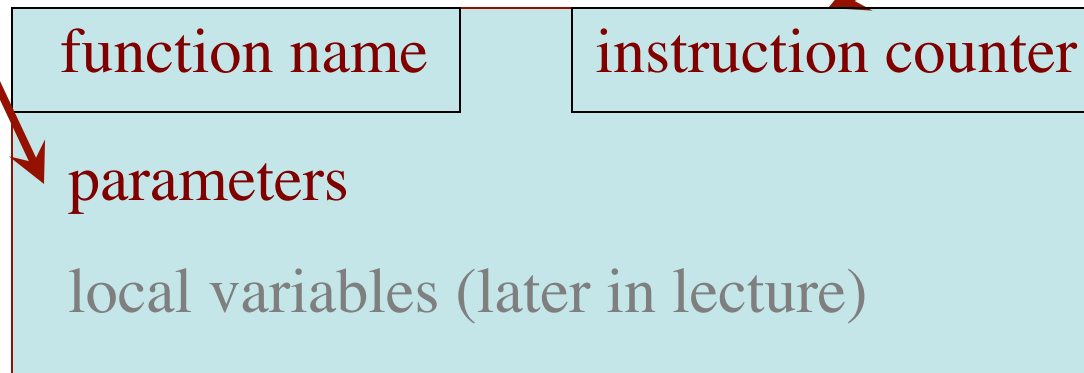
How Do Functions Work?

Draw template on
a piece of paper

- **Function Frame**: Representation of function call
- A **conceptual model** of Python

Draw parameters
as variables
(named boxes)

- Number of statement in the
function body to execute next
- **Starts with 1**



Text (Section 3.10) vs. Class

Textbook

to_centrigrade

$x \rightarrow 50.0$

This Class

to_centrigrade

1

x 50.0

Definition:

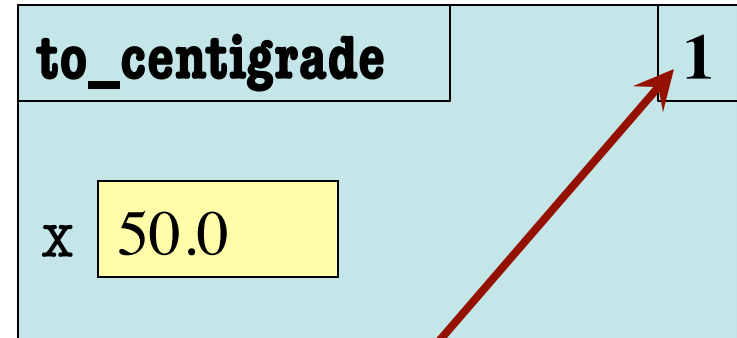
```
def to_centrigrade(x):  
    | return 5*(x-32)/9.0
```

Call: to_centrigrade(50.0)

Example: to_centigrade(50.0)

1. Draw a frame for the call
2. Assign the argument value to the parameter (in frame)
3. Execute the function body
 - Look for variables in the frame
 - If not there, look for global variables with that name
4. Erase the frame for the call

Initial call frame
(before exec body)



next line to execute

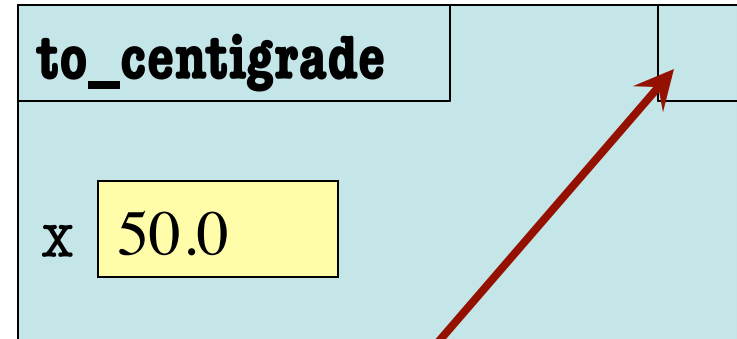
```
def to_centigrade(x):  
1 | return 5*(x-32)/9.0
```

Example: to_centigrade(50.0)

1. Draw a frame for the call
2. Assign the argument value to the parameter (in frame)
3. Execute the function body
 - Look for variables in the frame
 - If not there, look for global variables with that name
4. Erase the frame for the call

```
1  def to_centigrade(x):  
    |   return 5*(x-32)/9.0
```

Executing the
return statement



The return terminates;
no next line to execute

Example: to_centigrade(50.0)

1. Draw a frame for the call
2. Assign the argument value to the parameter (in frame)
3. Execute the function body
 - Look for variables in the frame
 - If not there, look for global variables with that name
4. Erase the frame for the call

ERASE WHOLE FRAME

```
1  def to_centigrade(x):  
    |   return 5*(x-32)/9.0
```

But don't actually
erase on an exam

Call Frames vs. Global Variables

- This does not work:

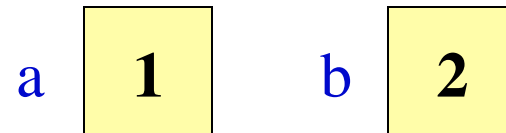
```
def swap(a,b):  
    """Swap vars a & b"""  
1    tmp = a  
2    a = b  
3    b = tmp
```

```
>>> a = 1
```

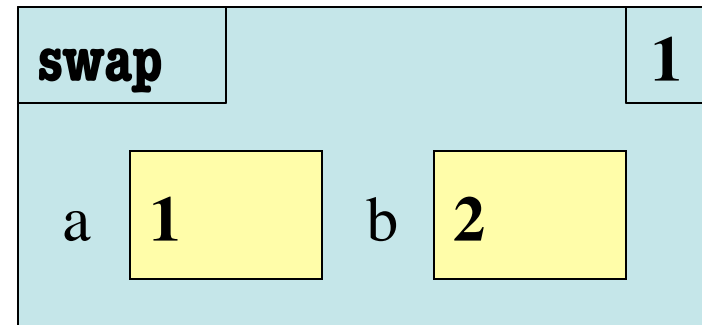
```
>>> b = 2
```

```
>>> swap(a,b)
```

Global Variables



Call Frame



Call Frames vs. Global Variables

- This does not work:

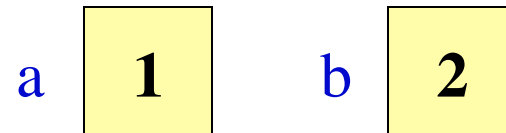
```
def swap(a,b):  
    """Swap vars a & b"""  
1   tmp = a  
2   a = b  
3   b = tmp
```

```
>>> a = 1
```

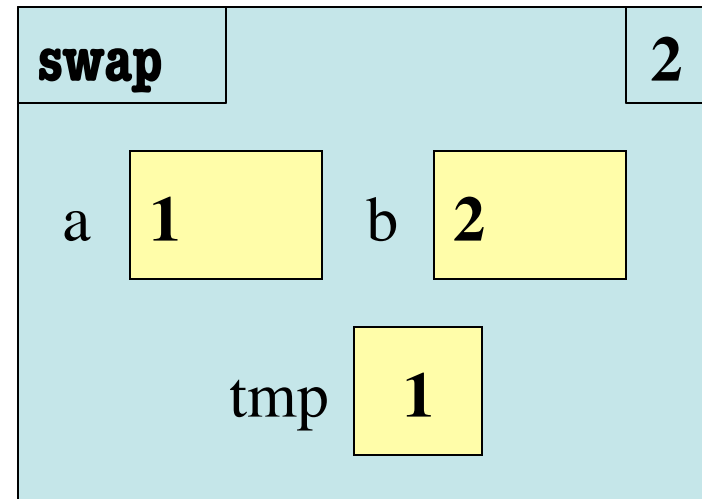
```
>>> b = 2
```

```
>>> swap(a,b)
```

Global Variables



Call Frame



Call Frames vs. Global Variables

- This does not work:

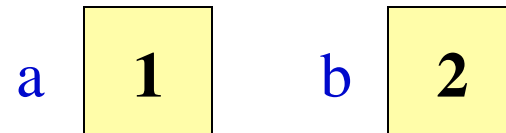
```
def swap(a,b):  
    """Swap vars a & b"""  
1    tmp = a  
2    a = b  
3    b = tmp
```

```
>>> a = 1
```

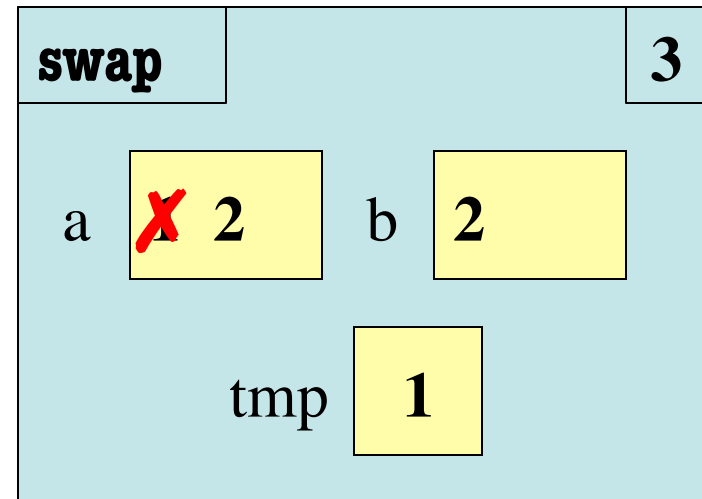
```
>>> b = 2
```

```
>>> swap(a,b)
```

Global Variables



Call Frame



Call Frames vs. Global Variables

- This does not work:

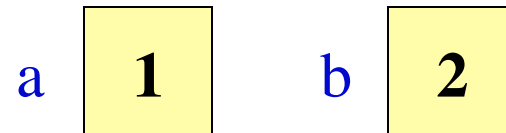
```
def swap(a,b):  
    """Swap vars a & b"""  
1   tmp = a  
2   a = b  
3   b = tmp
```

```
>>> a = 1
```

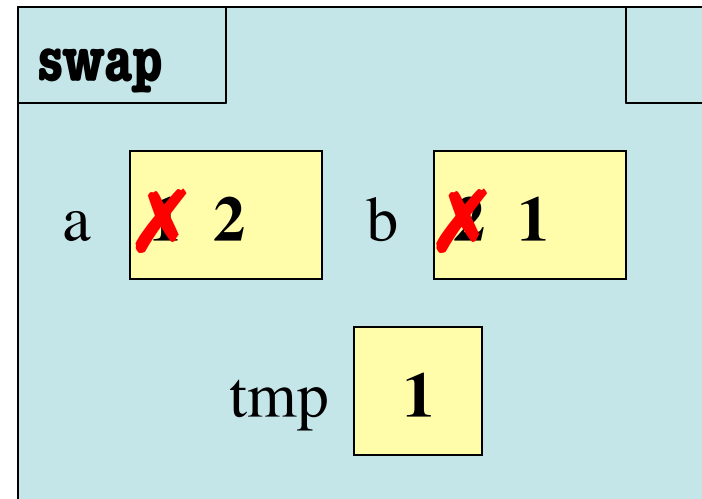
```
>>> b = 2
```

```
>>> swap(a,b)
```

Global Variables



Call Frame



Call Frames vs. Global Variables

- This does not work:

```
def swap(a,b):  
    """Swap vars a & b"""  
1   tmp = a  
2   a = b  
3   b = tmp
```

```
>>> a = 1
```

```
>>> b = 2
```

```
>>> swap(a,b)
```

Global Variables

a	1	b	2
---	---	---	---

Call Frame

Call Frames vs. Global Variables

- This does not work:

```
def swap(a,b):  
    """Swap vars a & b"""  
1   tmp = a  
2   a = b  
3   b = tmp
```

```
>>> a = 1
```

```
>>> b = 2
```

```
>>> swap(a,b)
```

Global Variables

a 1 b 2

Call Frame

See the **Python Tutor**
on course website

Recall: The Python API

The screenshot shows the Python documentation for `math.ceil(x)`. A green callout bubble points to the function name `math.ceil(x)`. Another green callout bubble points to the text "Return the ceiling of x as a float, the smallest integer value greater than or equal to x.", which is the function's docstring. A third green callout bubble points to the text "The following functions are provided by this module. Except when explicitly noted otherwise, all", which is part of the module's documentation. The screenshot also shows the "Table Of Contents" and "Next topic" sections.

Function name

Number of arguments

What the function evaluates to

- This is a **specification**
 - Enough info to use func.
 - But not how to implement
- Write them as **docstrings**

Anatomy of a Specification

```
def greet(n):
```

```
    """Prints a greeting to the name n
```

```
    Greeting has format 'Hello <n>!'
    Followed by a conversation starter.
```

```
    Precondition: n is a string
    representing a person's name"""
```

```
    print 'Hello '+n+'!'
```

```
    print 'How are you?'
```

One line description,
followed by blank line

More detail about the
function. It may be
many paragraphs.

Precondition specifies
assumptions we make
about the arguments

Anatomy of a Specification

```
def to_centigrade(x):
```

```
    """Returns: x converted to centigrade
```

```
    Value returned has type float.
```

```
    Precondition: x is a float measuring  
    temperature in fahrenheit"""
```

```
return 5*(x-32)/9.0
```

“Returns” indicates a fruitful function

More detail about the function. It may be many paragraphs.

Precondition specifies assumptions we make about the arguments

Preconditions

- Precondition is a **promise**
 - If precondition is true, the function works
 - If precondition is false, no guarantees at all
- Get **software bugs** when
 - Function precondition is not documented properly
 - Function is used in ways that violates precondition

```
>>> to_centrigrade(32)
```

```
0.0
```

```
>>> to_centrigrade(212)
```

```
100.0
```

```
>>> to_centrigrade('32')
```

```
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
```

```
File "temperature.py", line 19 ...
```

```
TypeError: unsupported operand type(s)  
for -: 'str' and 'int'
```

Precondition violated