LIST COMPREHENSION, FUNCTIONS AS OBJECTS, TESTING, DEBUGGING

(download slides and .py files to follow along)

6.100L Lecture 12

Ana Bell

LIST COMPREHENSIONS

LIST COMPREHENSIONS

- Applying a **function to every element of a sequence**, then creating a new list with these values is a common concept
- Example:

New list

Function to apply

```
def f(L):
    Lnew = []
    for e in L:
        Lnew.append(e**2)
    return Lnew
```

- Python provides a concise one-liner way to do this, called a **list comprehension**
  - Creates a new list
  - Applies a function to every element of another iterable
  - Optional, only apply to elements that satisfy a test

```
[expression for elem in iterable if test]
```

LIST COMPREHENSIONS

- Create a **new** list, by applying a function to every element of another iterable that satisfies a test

```
def f(L):
  Lnew = []
  for e in L:
    Lnew.append(e**2)
  return Lnew
```

New list

Look at every element

Function to apply

```
Lnew = [e**2 for e in L]
```

New list

LIST COMPREHENSIONS

- Create a **new** list, by applying a function to every element of another iterable that satisfies a test

```
def f(L):
  Lnew = []
  for e in L:
    Lnew.append(e**2)
  return Lnew
```

```
Lnew = [e**2 for e in L]
```

New list

Function to apply only if test is true

```
def f(L):
  Lnew = []
  for e in L:
    if e%2==0:
      Lnew.append(e**2)
  return Lnew
```

LIST COMPREHENSIONS

- Create a **new** list, by applying a function to every element of another iterable that satisfies a test

```
def f(L):
  Lnew = []
  for e in L:
    Lnew.append(e**2)
  return Lnew
```

```
Lnew = [e**2 for e in L]
```

```
def f(L):
  Lnew = []
  for e in L:
    if e%2==0:
      Lnew.append(e**2)
  return Lnew
```

New list

```
Lnew = [e**2 for e in L if e%2==0]
```

LIST COMPREHENSIONS

- Create a **new** list, by applying a function to every element of another iterable that satisfies a test

```
def f(L):
  Lnew = []
  for e in L:
    Lnew.append(e**2)
  return Lnew
```

```
Lnew = [e**2 for e in L]
```

```
def f(L):
  Lnew = []
  for e in L:
    if e%2==0:
      Lnew.append(e**2)
  return Lnew
```

Loop over elements

```
Lnew = [e**2 for e in L if e%2==0]
```

LIST COMPREHENSIONS

- Create a **new** list, by applying a function to every element of another iterable that satisfies a test

```
def f(L):
  Lnew = []
  for e in L:
    Lnew.append(e**2)
  return Lnew
```

```
Lnew = [e**2 for e in L]
```

```
def f(L):
  Lnew = []
  for e in L:
    if e%2==0:
      Lnew.append(e**2)
  return Lnew
```

Function to apply only if test is true

```
Lnew = [e**2 for e in L if e%2==0]
```

LIST COMPREHENSIONS

[*expression* for *elem* in *iterable* if *test*]

- This is equivalent to invoking this function (where expression is a function that computes that expression)

```
def f(expr, old_list, test = lambda x: True):
    new_list = []
    for e in old_list:
        if test(e):
            new_list.append(expr(e))
    return new_list
```

```
[e**2 for e in range(6)]              → [0, 1, 4, 9, 16, 25]
[e**2 for e in range(8) if e%2 == 0]  → [0, 4, 16, 36]
[[e,e**2] for e in range(4) if e%2 != 0]  → [[1,1], [3,9]]
```

YOU TRY IT!

- What is the value returned by this expression?
  - Step1: what are **all values** in the sequence
  - Step2: which **subset of values** does the condition filter out?
  - Step3: **apply the function** to those values

```
[len(x) for x in ['xy', 'abcd', 7, '4.0'] if type(x) == str]
```

FUNCTIONS: DEFAULT PARAMETERS

SQUARE ROOT with BISECTION

```
def bisection_root(x):
    epsilon = 0.01
    low = 0
    high = x
    guess = (high + low)/2.0
    while abs(guess**2 - x) >= epsilon:
        if guess**2 < x:
            low = guess
        else:
            high = guess
        guess = (high + low)/2.0
    return guess

print(bisection_root(123))
```

ANOTHER PARAMETER

- Motivation: want a more accurate answer
  `def bisection_root(x)` can be improved
- Options?
  - Change epsilon **inside function** (all function calls are affected)
  - Use an epsilon **outside function** (global variables are bad)
  - Add epsilon as **an argument** to the function

epsilon as a PARAMETER

```
def bisection_root(x, epsilon):
    low = 0
    high = x
    guess = (high + low)/2.0
    while abs(guess**2 - x) >= epsilon:
        if guess**2 < x:
            low = guess
        else:
            high = guess
        guess = (high + low)/2.0
    return guess

print(bisection_root(123, 0.01))
```

KEYWORD PARAMETERS & DEFAULT VALUES

`def bisection_root(x, epsilon)` can be improved

- We added epsilon as an argument to the function
  - **Most of the time** we want some **standard value**, 0.01
  - **Sometimes**, we may want to use some **other value**
- Use a keyword parameter aka a **default parameter**

Epsilon as a KEYWORD PARAMETER

```
def bisection_root(x, epsilon=0.01):
    low = 0
    high = x
    guess = (high + low)/2.0
    while abs(guess**2 - x) >= epsilon:
        if guess**2 < x:
            low = guess
        else:
            high = guess
        guess = (high + low)/2.0
    return guess

print(bisection_root(123))
print(bisection_root(123, 0.5))
```

Default parameter, with default value of 0.01

Uses epsilon as 0.01 (the default one in function def)

Uses epsilon as 0.5

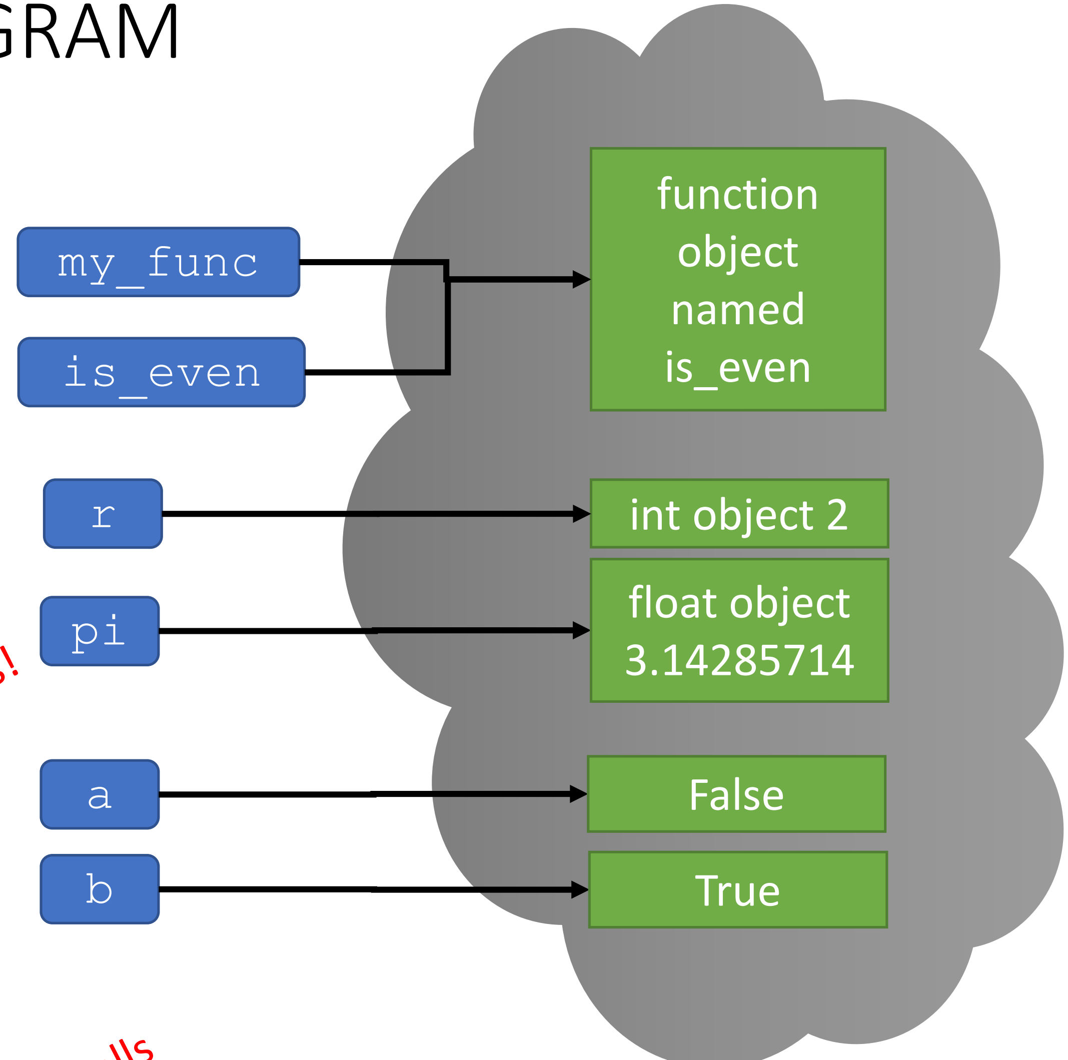
RULES for KEYWORD PARAMETERS

- In the **function definition**:
  - Default parameters must go at the end

- These are **ok for calling a function**:
  - `bisection_root_new(123)`
  - `bisection_root_new(123, 0.001)`
  - `bisection_root_new(123, epsilon=0.001)`
  - `bisection_root_new(x=123, epsilon=0.1)`
  - `bisection_root_new(epsilon=0.1, x=123)`
- These are **not ok for calling a function**:
  - `bisection_root_new(epsilon=0.001, 123) #error`
  - `bisection_root_new(0.001, 123) #no error but wrong`

FUNCTIONS RETURNING FUNCTIONS

OBJECTS IN A PROGRAM




```
def is_even(i):
    return i%2 == 0

r = 2

pi = 22/7

my_func = is_even

a = is_even(3)

b = my_func(4)
```

NOT a function call, just names!

Function calls

FUNCTIONS CAN RETURN FUNCTIONS

```
def make_prod(a):
    def g(b):
        return a*b
    return g
```

This function def is inside another function.

This is NOT a function call!

```
val = make_prod(2)(3)
print(val)
```

SAME

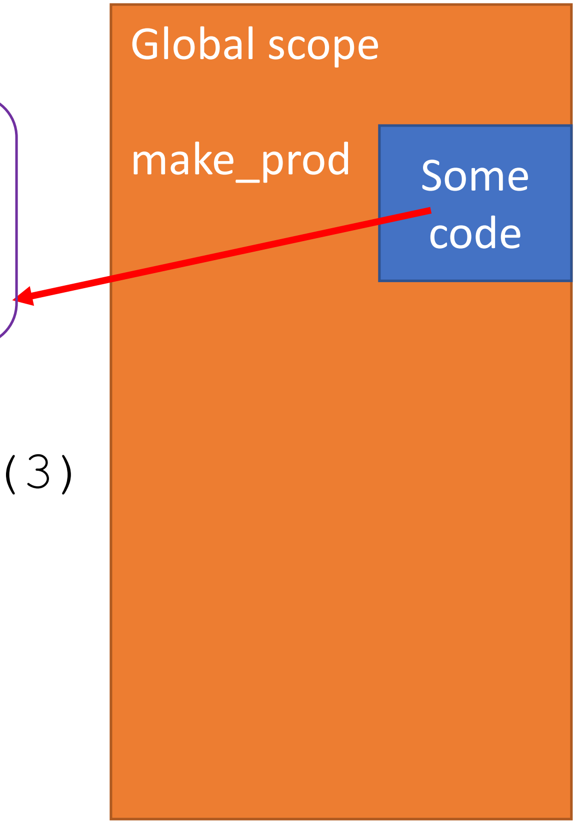
```
doubler = make_prod(2)
val = doubler(3)
print(val)
```

SCOPE DETAILS FOR WAY 1

```
def make_prod(a):
    def g(b):
        return a*b
    return g

val = make_prod(2)(3)
print(val)
```

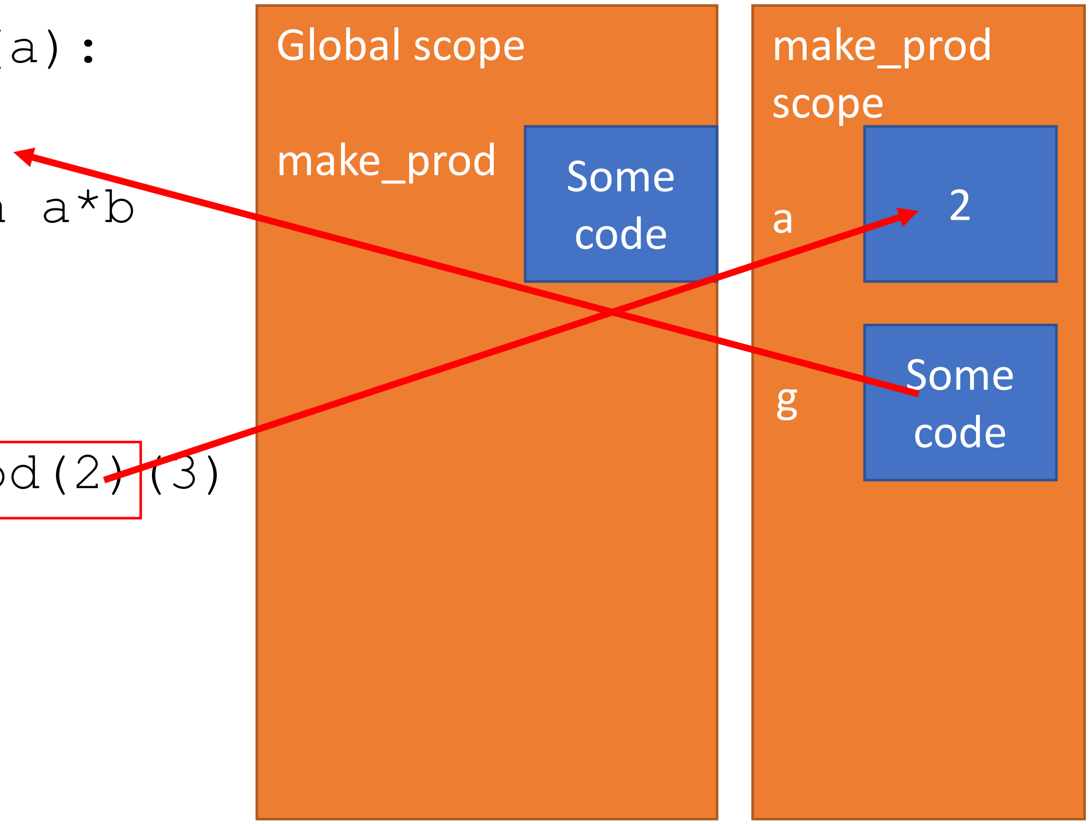
SCOPE DETAILS FOR WAY 1

```
def make_prod(a):
    def g(b):
        return a*b
    return g

val = make_prod(2)(3)
print(val)
```



SCOPE DETAILS FOR WAY 1

```
def make_prod(a):
    def g(b):
        return a*b
    return g

val = make_prod(2)(3)
print(val)
```



NOTE: definition of g is done within scope of make_prod, so binding of g is within that frame/scope

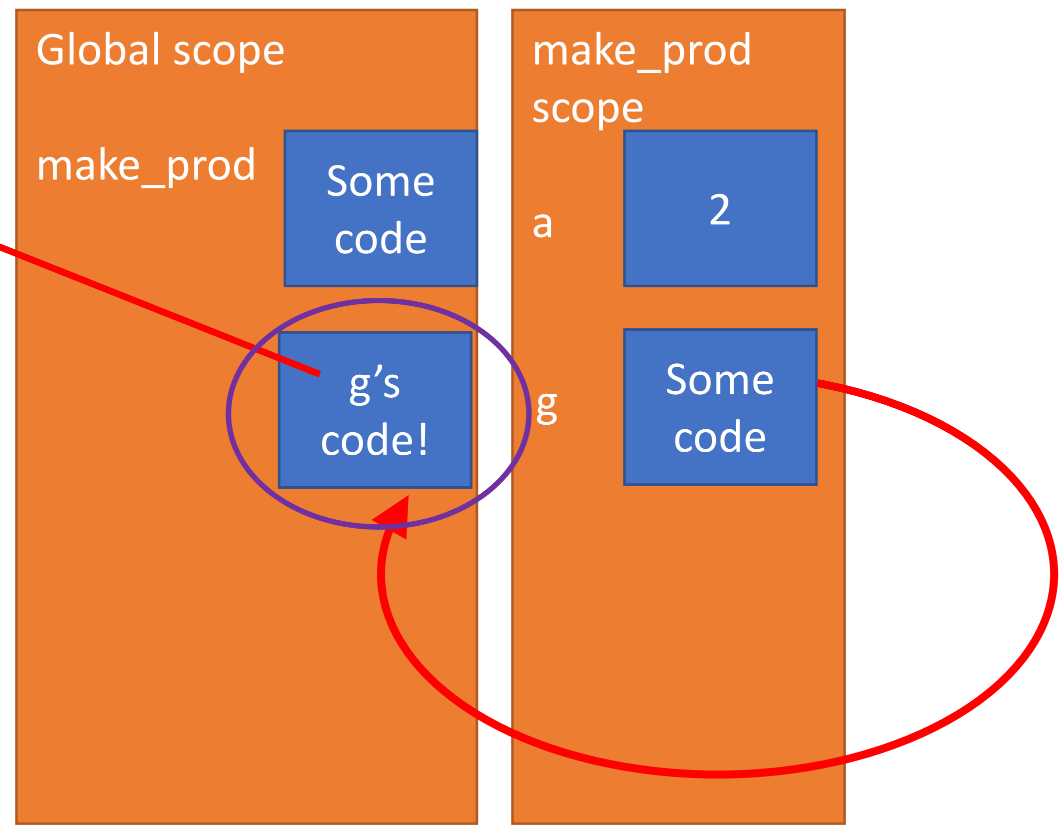
Since g is bound in this frame, cannot access it by evaluation in global frame

g can only be accessed within call to make_prod, and each call will create a new, internal g

SCOPE DETAILS FOR WAY 1

```
def make_prod(a):
    def g(b):
        return a*b
    return g

val = make_prod(2)(3)
print(val)
```

This is g



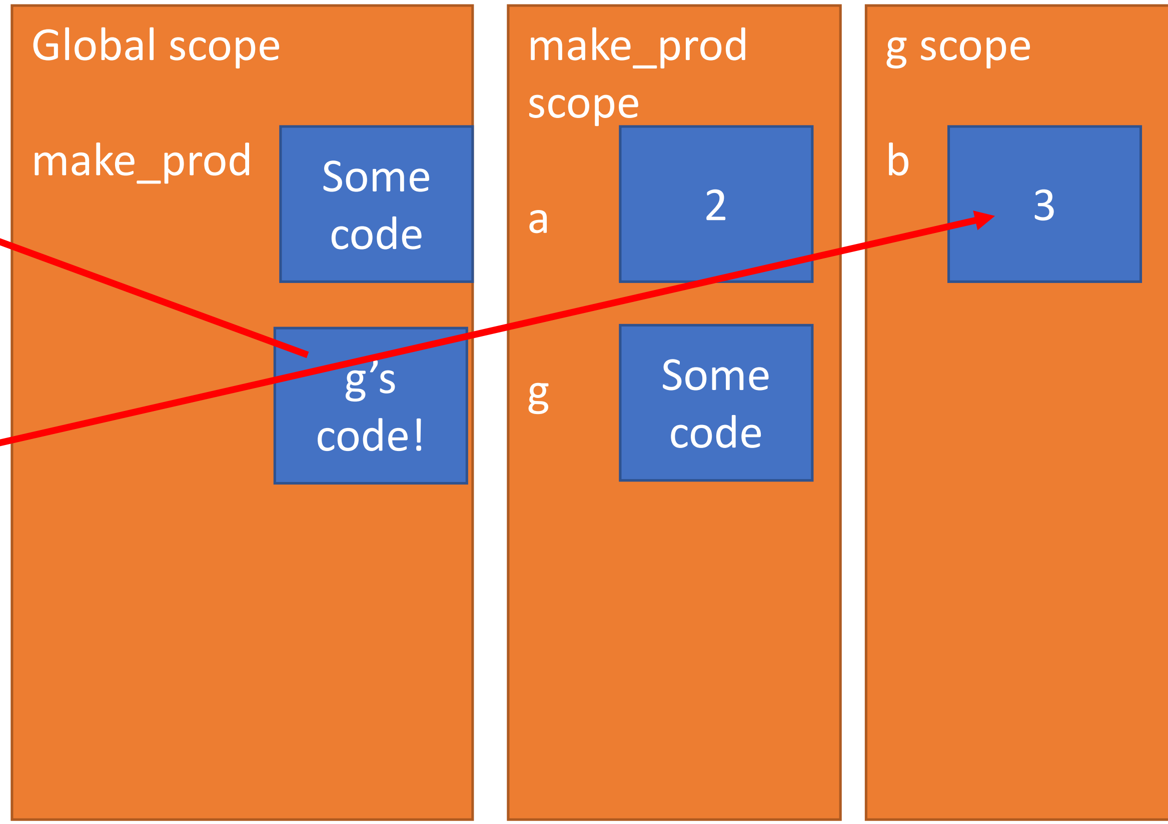
Evaluating `make_prod(2)` has returned an anonymous procedure

Returns pointer to g code

SCOPE DETAILS FOR WAY 1

```
def make_prod(a):
    def g(b):
        return a*b
    return g


val = make_prod(2)(3)
print(val)
```

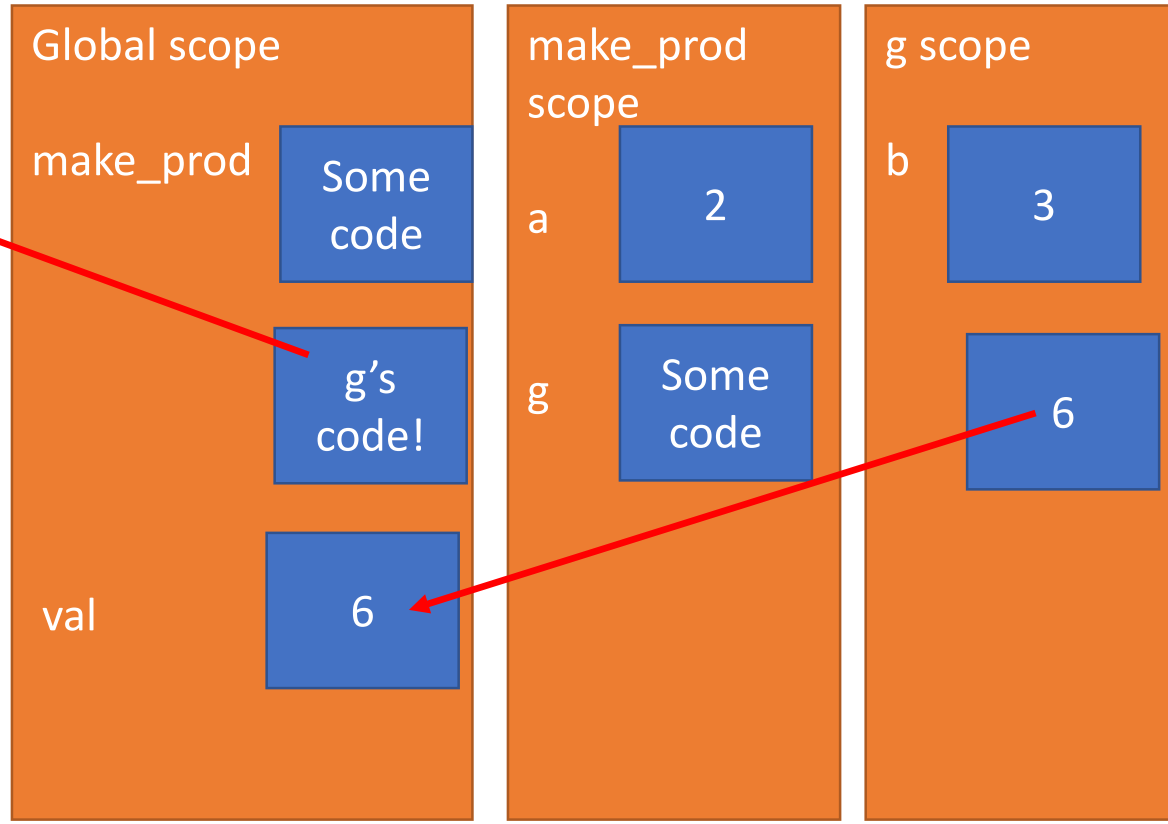
Call is g(3)



SCOPE DETAILS FOR WAY 1

```
def make_prod(a):
    def g(b):
        return a*b
    return g

val = make_prod(2)(3)
print(val)
```

Internal procedure only accessible within scope from parent procedure's call



How does g get value for a?

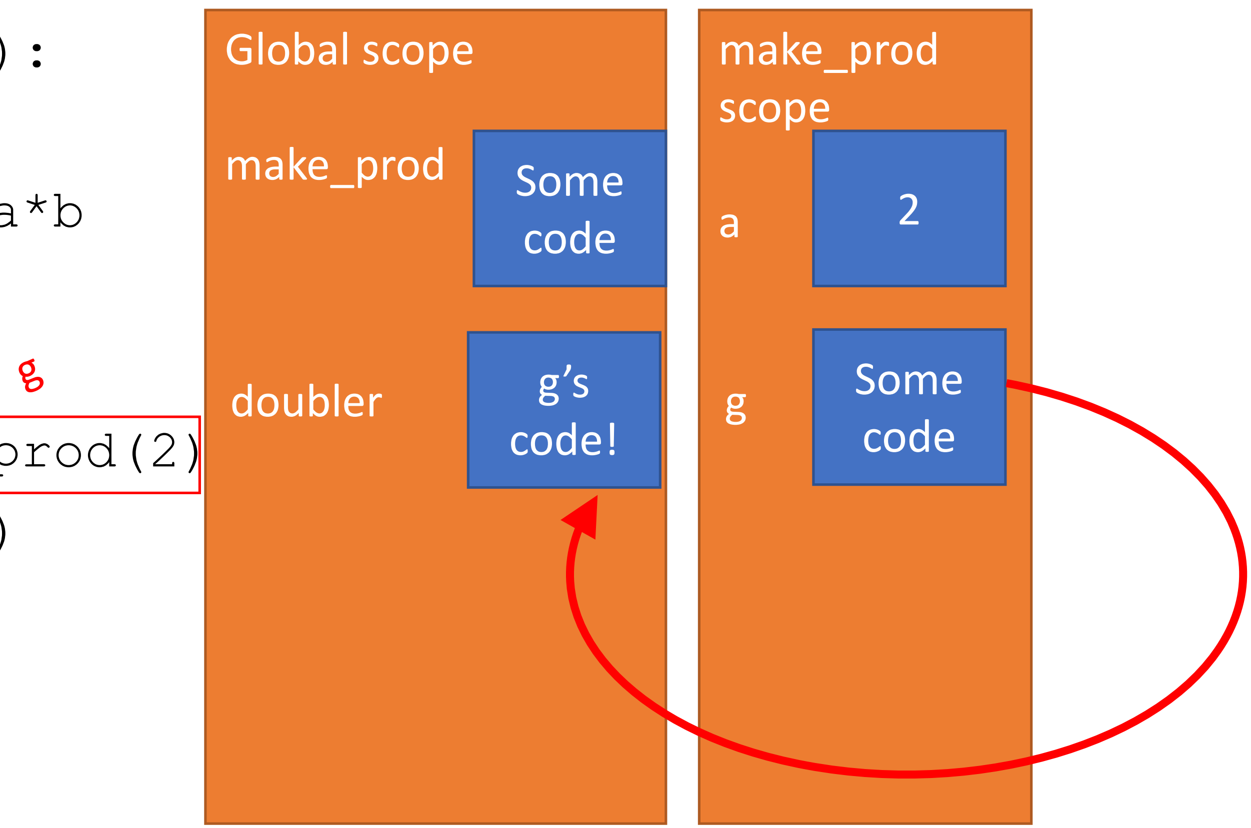
Interpreter can move up hierarchy of frames to see both b and a values

SCOPE DETAILS FOR WAY 2

```
def make_prod(a):
    def g(b):
        return a*b
    return g

doubler = make_prod(2)
val = doubler(3)
print(val)
```

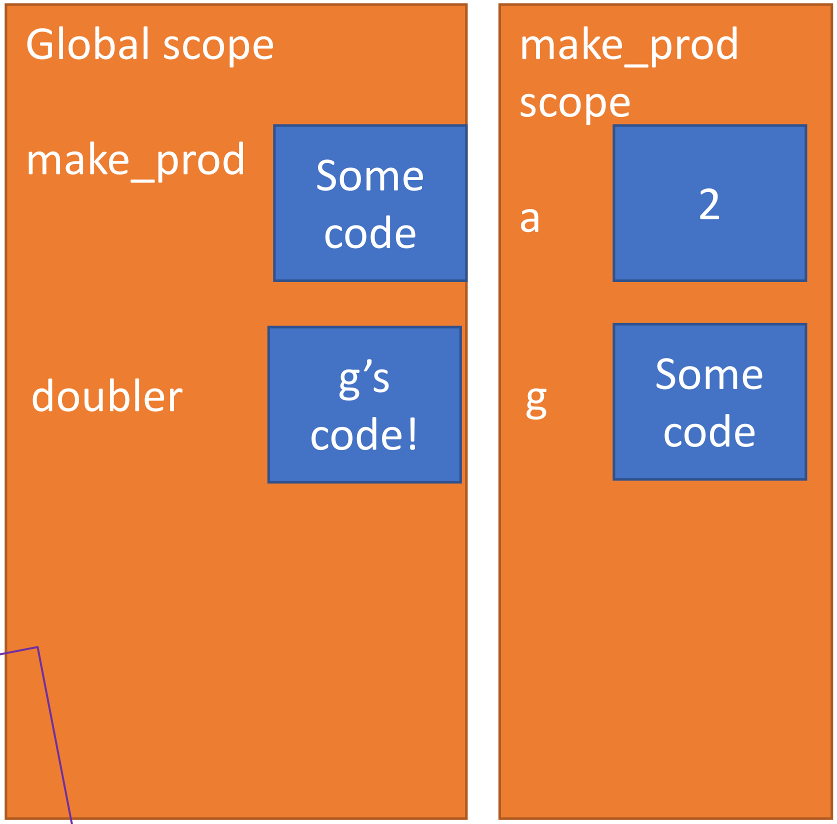
SCOPE DETAILS FOR WAY 2

```
def make_prod(a):
    def g(b):
        return a*b
    return g

doubler = make_prod(2)
val = doubler(3)
print(val)
```



SCOPE DETAILS FOR WAY 2

```
def make_prod(a):
    def g(b):
        return a*b
    return g

doubler = make_prod(2)
val = doubler(3)
print(val)
```

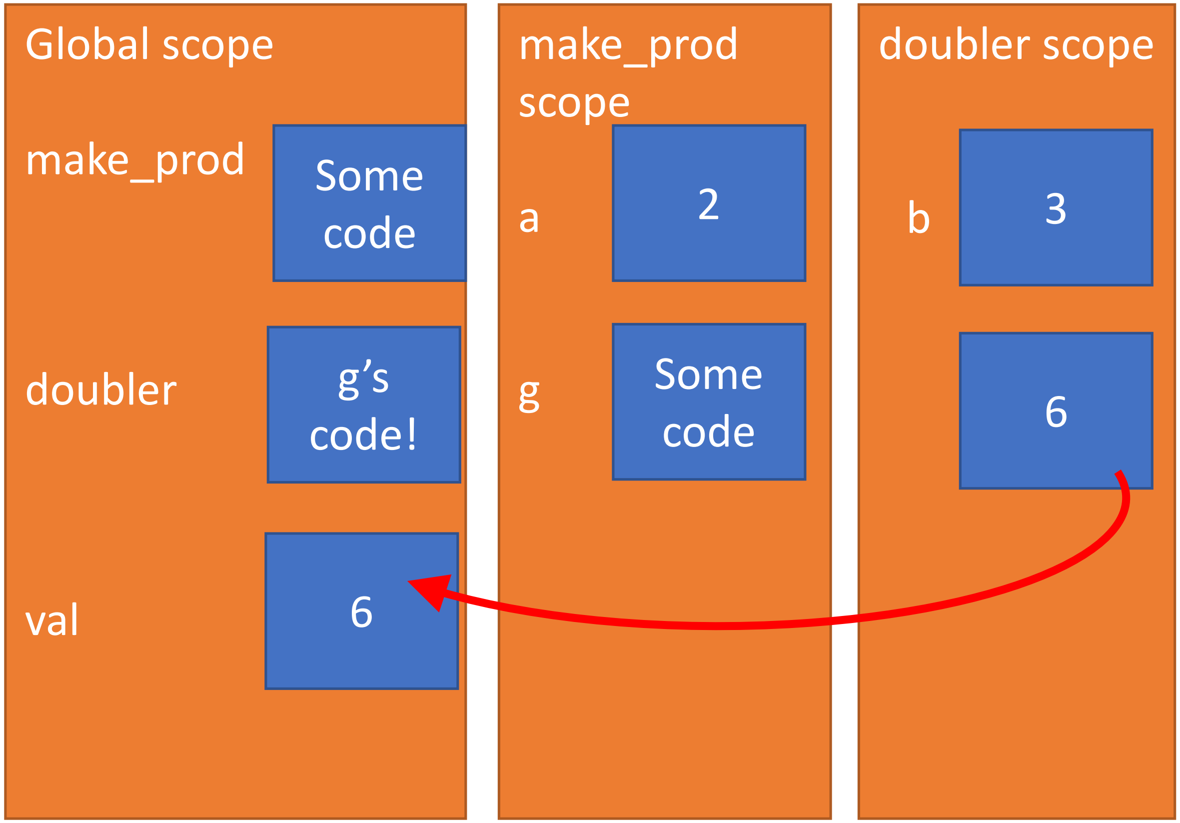


Evaluating `make_prod(2)` has same effect as previous example – doubler is a **function object**

SCOPE DETAILS FOR WAY 2

```
def make_prod(a):
    def g(b):
        return a*b
    return g

doubler = make_prod(2)
val = doubler(3)
print(val)
```

Now invoking g(3)



Returns value

WHY BOTHER RETURNING FUNCTIONS?

- Code can be **rewritten** without returning function objects
- Good software design
  - Embracing ideas of **decomposition**, **abstraction**
  - Another **tool** to structure code
- Interrupting execution
  - Example of **control flow**
  - A way to achieve **partial execution** and use result somewhere else before finishing the full evaluation

TESTING and DEBUGGING

DEFENSIVE PROGRAMMING

- Write **specifications** for functions
- **Modularize** programs
- Check **conditions** on inputs/outputs (assertions)

TESTING/VALIDATION

- **Compare** input/output pairs to specification
- “It’s not working!”
- “How can I break my program?”

DEBUGGING

- **Study events** leading up to an error
- “Why is it not working?”
- “How can I fix my program?”

SET YOURSELF UP FOR EASY TESTING AND DEBUGGING

- From the **start**, design code to ease this part
- Break program up into **modules** that can be tested and debugged individually
- **Document constraints** on modules
  - What do you expect the input to be?
  - What do you expect the output to be?
- **Document assumptions** behind code design

WHEN ARE YOU READY TO TEST?

- Ensure **code runs**
  - Remove syntax errors
  - Remove static semantic errors
  - Python interpreter can usually find these for you
- Have a **set of expected results**
  - An input set
  - For each input, the expected output

CLASSES OF TESTS

- **Unit testing**
  - Validate each piece of program
  - **Testing each function** separately
- **Regression testing**
  - Add test for bugs as you find them
  - **Catch reintroduced** errors that were previously fixed
- **Integration testing**
  - Does **overall program** work?
  - Tend to rush to do this

TESTING APPROACHES

- **Intuition** about natural boundaries to the problem

```
def is_bigger(x, y):
    """ Assumes x and y are ints
    Returns True if y is less than x, else False """
```

  - can you come up with some natural partitions?
- If no natural partitions, might do **random testing**
  - Probability that code is correct increases with more tests
  - Better options below
- **Black box testing**
  - Explore paths through specification
- **Glass box testing**
  - Explore paths through code

BLACK BOX TESTING

```
def sqrt(x, eps):
    """ Assumes x, eps floats, x >= 0, eps > 0
    Returns res such that x-eps <= res*res <= x+eps """
```

- Designed **without looking** at the code
- Can be done by someone other than the implementer to avoid some implementer **biases**
- Testing can be **reused** if implementation changes
- **Paths** through specification
    - Build test cases in different natural space partitions
    - Also consider boundary conditions (empty lists, singleton list, large numbers, small numbers)

BLACK BOX TESTING

```
def sqrt(x, eps):
    """ Assumes x, eps floats, x >= 0, eps > 0
    Returns res such that x-eps <= res*res <= x+eps """
```

CASE	x	eps
boundary	0	0.0001
perfect square	25	0.0001
less than 1	0.05	0.0001
irrational square root	2	0.0001
extremes	2	1.0/2.0**64.0
extremes	1.0/2.0**64.0	1.0/2.0**64.0
extremes	2.0**64.0	1.0/2.0**64.0
extremes	1.0/2.0**64.0	2.0**64.0
extremes	2.0**64.0	2.0**64.0

GLASS BOX TESTING

- **Use code** directly to guide design of test cases
- Called **path-complete** if every potential path through code is tested at least once
- What are some **drawbacks** of this type of testing?
  - Can go through loops arbitrarily many times
  - Missing paths
- Guidelines
  - Branches → exercise all parts of a conditional
  - For loops → loop not entered; body of loop executed exactly once; body of loop executed more than once
  - While loops → same as for loops, cases that catch all ways to exit loop

GLASS BOX TESTING

```
def abs(x):
    """ Assumes x is an int
    Returns x if x>=0 and -x otherwise """
    if x < -1:
        return -x
    else:
        return x
```

- Aa path-complete test suite could **miss a bug**
- Path-complete test suite: 2 and -2
- But abs(-1) incorrectly returns -1
- Should still test boundary cases

DEBUGGING

- Once you have discovered that your code does not run properly, you want to:
  - Isolate the bug(s)
  - Eradicate the bug(s)
  - Retest until code runs correctly for all cases
  - Steep learning curve
- Goal is to have a bug-free program
- Tools
  - **Built in** to IDLE and Anaconda
  - **Python Tutor**
  - `print` statement
  - Use your brain, be **systematic** in your hunt

ERROR MESSAGES – EASY

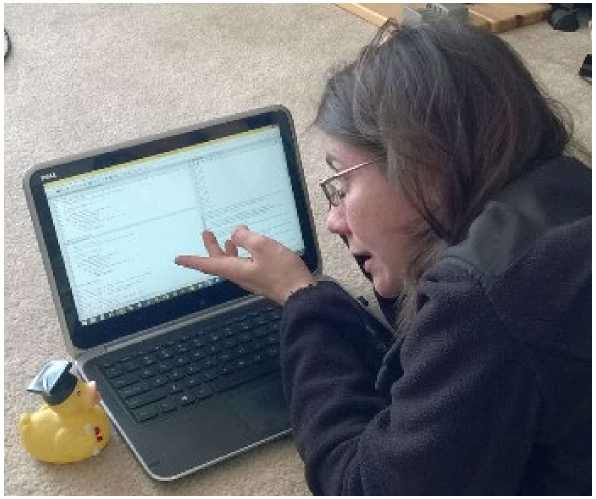
- Trying to access beyond the limits of a list
  `test = [1,2,3]` then `test[4]` → `IndexError`
- Trying to convert an inappropriate type
  `int(test)` → `TypeError`
- Referencing a non-existent variable
  `a` → `NameError`
- Mixing data types without appropriate coercion
  `'3'/4` → `TypeError`
- Forgetting to close parenthesis, quotation, etc.

```
a = len([1,2,3]
print(a)
```

→ `SyntaxError`

LOGIC ERRORS - HARD

- **think** before writing new code
- **draw** pictures, take a break
- **explain** the code to
  - someone else
  - a rubber ducky



DEBUGGING STEPS

- **Study** program code
  - Don't ask what is wrong
  - Ask how did I get the unexpected result
  - Is it part of a family?
- **Scientific method**
  - Study available data
  - Form hypothesis
  - Repeatable experiments
  - Pick simplest input to test with

PRINT STATEMENTS

- Good way to **test hypothesis**
- When to print
  - Enter function
  - Parameters
  - Function results
- Use **bisection method**
  - Put print halfway in code
  - Decide where bug may be depending on values