Lecture 9: Abstraction, Decomposition, Functions

Comp 102

Forman Christian University

Recap

So far....

- We've seen **program construction** i.e **building blocks**
  - Data types, Expressions
  - Conditions, Loops

So far....

- We've seen **program construction** i.e **building blocks**
  - Data types, Expressions
  - Conditions, Loops
- That's **ALL** you need to write **any program**

So far....

- We've seen **program construction** i.e **building blocks**
  - Data types, Expressions
  - Conditions, Loops
- That's **ALL** you need to write **any program**
- But writing **LARGE** programs is a completely different story

Now we'll ...

- See how to **organize** and **reuse** these building blocks

Now we'll ...

- See how to **organize** and **reuse** these building blocks
- Talk about ways of doing that using:
  - **Abstraction**
  - **Decomposition**
- Learn about Python **Functions**

Example: **The Smartphone**

A black box, and can be viewed in terms of:

- its inputs
- its outputs
- how outputs are related to inputs, without any knowledge of its internal workings
- implementation is "opaque" (or **black**)

The Smartphone Abstraction

- User **doesn't know the details** of how it works
  - We **don't need to know how something works** in order to know how to use it

The Smartphone Abstraction

- User **doesn't know the details** of how it works
  - We **don't need to know how something works** in order to know how to use it
- User **does know the interface**
  - Device converts a sequence of screen touches and sounds into expected useful functionality

The Smartphone Abstraction

- User **doesn't know the details** of how it works
  - We **don't need to know how something works** in order to know how to use it
- User **does know the interface**
  - Device converts a sequence of screen touches and sounds into expected useful functionality
- Know **relationship** between input and output

Abstraction

- **Hiding** the **details** of how something works
- **Showing** only **essential** parts
- **Simplifing** the **complexity** of a system

Abstraction

- **Hiding** the **details** of how something works
- **Showing** only **essential** parts
- **Simplifing** the **complexity** of a system

Most of the things in real life like Smartphone, Computer, Car, etc. are all examples of **Abstraction**

Abstraction enables **Decomposition**

- Breaking down a complex problem into **smaller, more manageable** parts
- Solving each part **separately**
- Combining the solutions to get the **final solution**

Big Idea

Apply **Abstraction** *(put in a black box)* and **Decomposition** *(split into self-contained parts)* to **Programming**

Abstraction

Hide Details with Abstraction

- In programming, want to think of piece of code as **black box**
  - **Hide tedious coding details** from the user
  - **Reuse** black box at different parts in the code (no copy/pasting!)

Hide Details with Abstraction

- In programming, want to think of piece of code as **black box**
  - **Hide tedious coding details** from the user
  - **Reuse** black box at different parts in the code (no copy/pasting!)
- **Coder creates details**, and designs interface
- **User does not need or want** to see details

Achieve **Abstraction** with a **Function**

Function lets us capture code in a **black box**

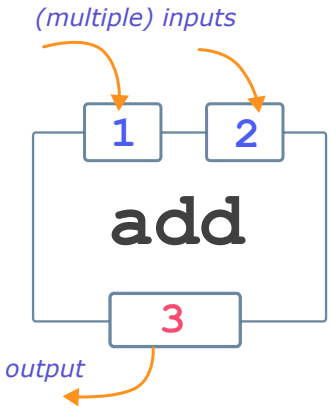
```
add(1,2)
```

```
len('comp 102')
```

```
max(5,1,4,3)
```

```
round(2.4)
```

Function as a Black Box



Creating Function Objects

```
def is_even(i):
    """
    input: i, a positive int
    returns True if i is even, otherwise False
    """
    if i%2 == 0:
        return True
    else:
        return False
```

Creating Function Objects

keyword

```
def is_even(i):
    """
    input: i, a positive int
    returns True if i is even, otherwise False
    """
    if i%2 == 0:
        return True
    else:
        return False
```

Creating Function Objects

keyword: def
name: is_even

```
def is_even(i):
    """
    input: i, a positive int
    returns True if i is even, otherwise False
    """
    if i%2 == 0:
        return True
    else:
        return False
```

Creating Function Objects

keyword name parameters

```
def is_even(i):
    """
    input: i, a positive int
    returns True if i is even, otherwise False
    """
    if i%2 == 0:
        return True
    else:
        return False
```

Creating Function Objects

keyword

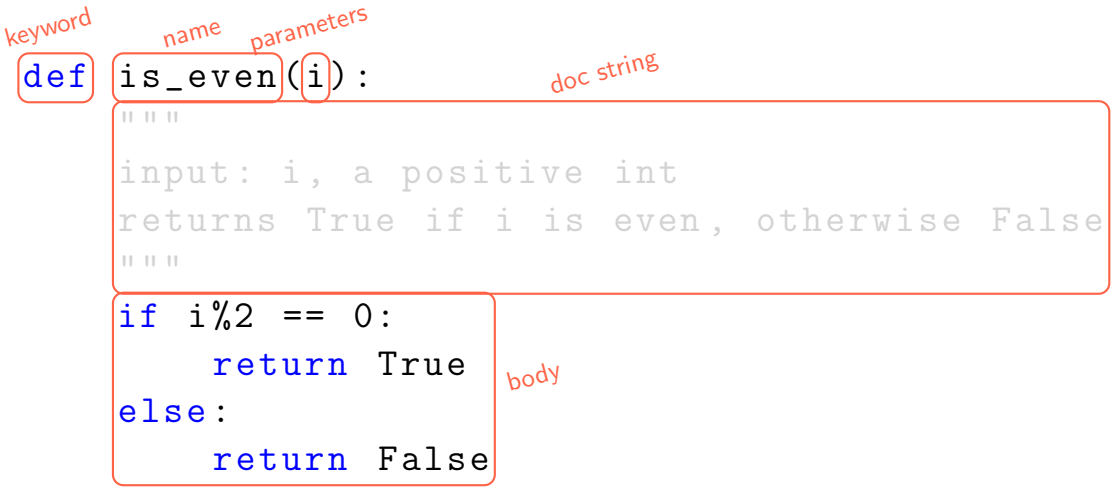
name

parameters

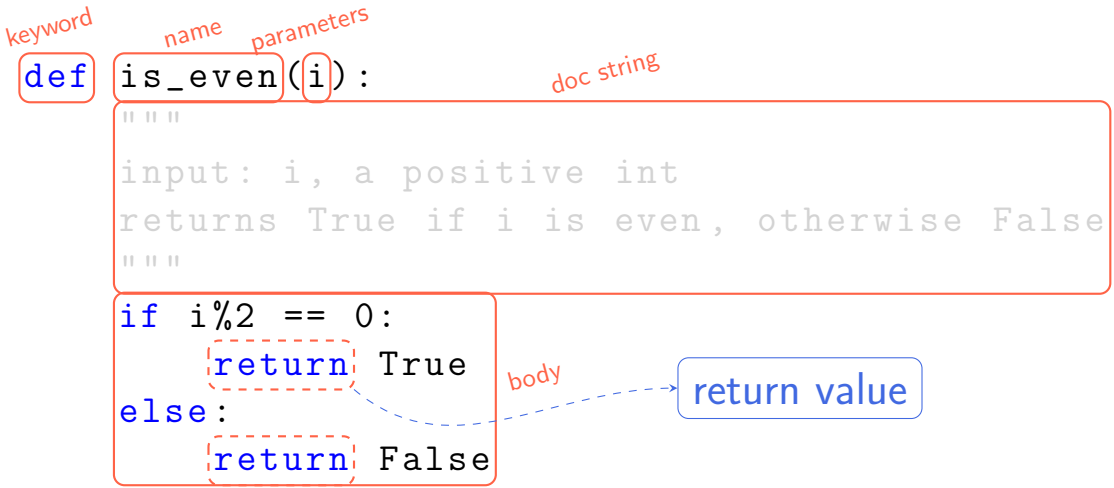
doc string

```
def is_even(i):
    """
    input: i, a positive int
    returns True if i is even, otherwise False
    """
    if i%2 == 0:
        return True
    else:
        return False
```

Creating Function Objects



Creating Function Objects



Creating Function Objects

- Can you make the code **cleaner**?
  - `i % 2` is a Boolean that evaluates to `True`/`False` already

```
def is_even( i ):
    """
    Input: i, a positive int
    Returns True if i is even, otherwise False
    """
    return i%2 == 0
```

print vs return: They Are NOT the Same!

Version 1: print

```
def is_even_v1(i):
    print(i%2 == 0)
```

Prints to console, gives back None

Version 2: return

```
def is_even_v2(i):
    return i%2 == 0
```

Sends value back to caller

print vs return: They Are NOT the Same!

Version 1: `print`

```
def is_even_v1(i):
    print(i%2 == 0)
```

Prints to console, gives back `None`

Version 2: `return`

```
def is_even_v2(i):
    return i%2 == 0
```

Sends value back to caller

```
a = is_even_v1(3)    # a is None
b = is_even_v2(3)    # b is False
```

print vs return: They Are NOT the Same!

Version 1: `print`

```
def is_even_v1(i):
    print(i%2 == 0)
```

Prints to console, gives back `None`

Version 2: `return`

```
def is_even_v2(i):
    return i%2 == 0
```

Sends value back to caller

```
a = is_even_v1(3)    # a is None
b = is_even_v2(3)    # b is False
```

a holds `None`

b holds `False`

What Happens Without return?

```
def greet(name):
    print("Hello", name)

x = greet("Ali")
print(x)
```

What Happens Without `return`?

```
def greet(name):
    print("Hello", name)

x = greet("Ali")
print(x)
```

Console output:

```
Hello Ali
None
```

What Happens Without `return`?

```
def greet(name):
    print("Hello", name)

x = greet("Ali")
print(x)
```

Console output:

```
Hello Ali
None
```

Every Python function returns something. If you don't write `return`, Python returns `None` automatically.

Big Idea

A function's code only runs when you call *(invoke)* the function.

Together in a File

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

is_even(3)
```

Together in a File

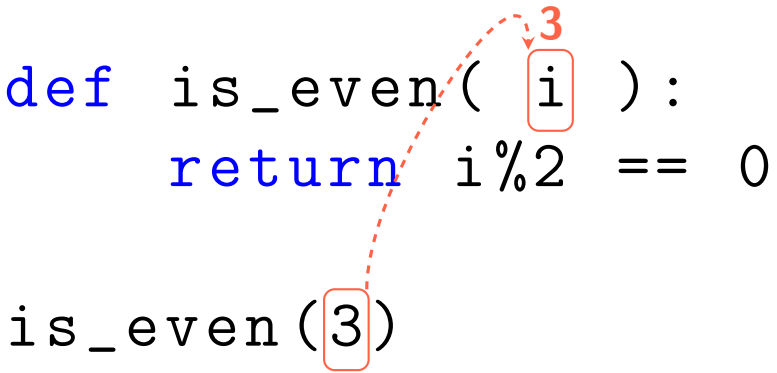
```
def is_even( i ):
    return i%2 == 0
```

function object

```
is_even(3)
```

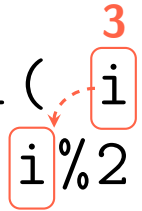
call expression

Together in a File



Together in a File

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

is_even(3)
```



Together in a File

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

is_even(3)
```

3

3

3%2==0

Together in a File

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

3%2==0

False

```
is_even(3)
```

Together in a File

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

```
False
```

You Try!

Write code that satisfies the following specs

```
def div_by(n, d):
    """ n and d are integers > 0
    Returns True if d divides n evenly
     and False otherwise """
```

Test your code with:

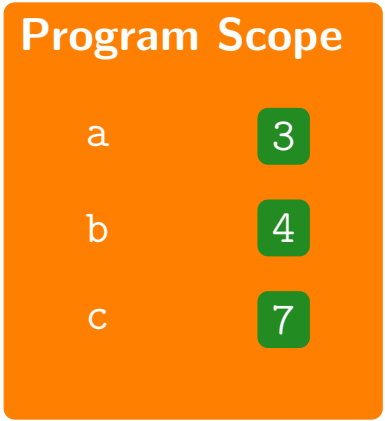
n = 10 and d = 3

n = 195 and d = 13

Program Scope

Program Scope

```
a = 3
b = 4
c = a+b
```



Program Scope

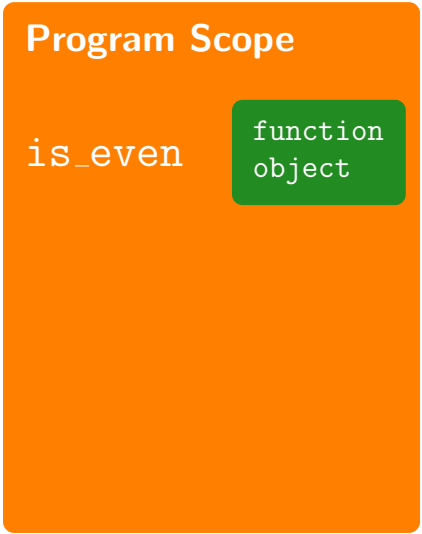
```
def is_even( i ):
    '''my black box'''
    print("inside is_even")
    return i%2 == 0

a = is_even(3)
b = is_even(10)
c = is_even(123456)
```

Program Scope

Program Scope

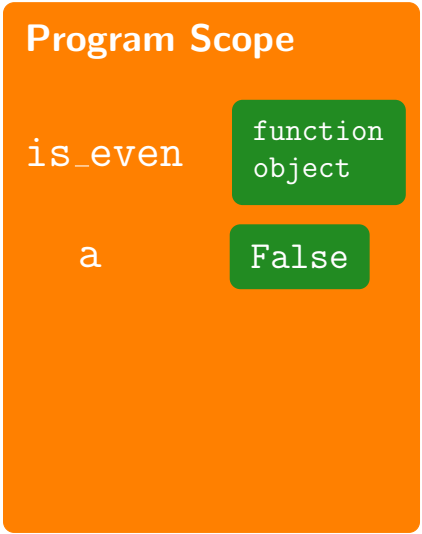
```
def is_even( i ):
    '''my black box'''
    print("inside is_even")
    return i%2 == 0
```

```
a = is_even(3)
b = is_even(10)
c = is_even(123456)
```



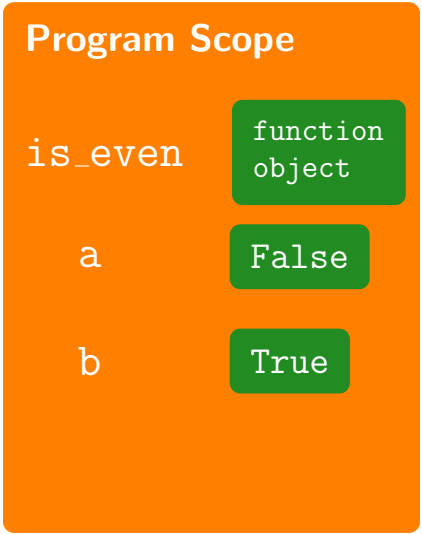
Program Scope

```
def is_even( i ):
    '''my black box'''
    print("inside is_even")
    return i%2 == 0

a = is_even(3)
b = is_even(10)
c = is_even(123456)
```



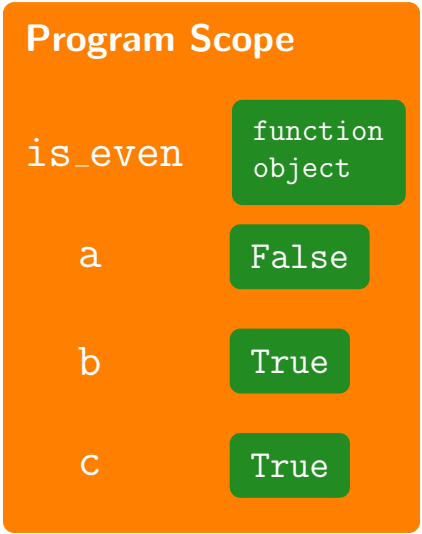
Program Scope

```
def is_even( i ):
    '''my black box'''
    print("inside is_even")
    return i%2 == 0

a = is_even(3)
b = is_even(10)
c = is_even(123456)
```



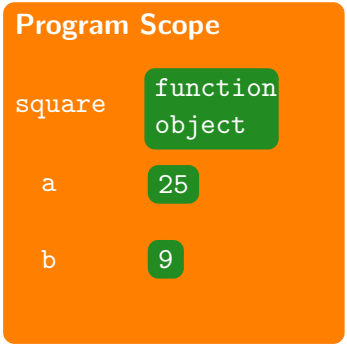
Program Scope

```
def is_even( i ):
    '''my black box'''
    print("inside is_even")
    return i%2 == 0

a = is_even(3)
b = is_even(10)
c = is_even(123456)
```



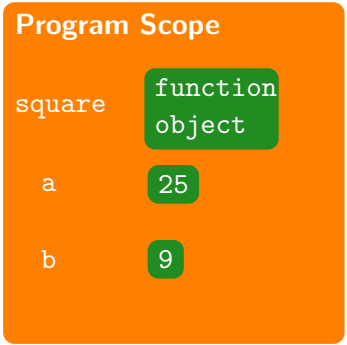
What Gets a Name in Program Scope?

```
def square(x):
    return x * x

a = square(5)
b = square(3)
```



What Gets a Name in Program Scope?

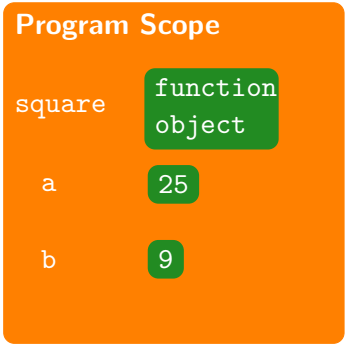
```
def square(x):
    return x * x

a = square(5)
b = square(3)
```



What about x? Is it in Program Scope?

What Gets a Name in Program Scope?

```
def square(x):
    return x * x

a = square(5)
b = square(3)
```



What about x? Is it in Program Scope?

No! We'll learn why next lecture.

Inserting Functions in Code

- Remember how expressions are replaced with the value?
- The **function call** is **replaced** with the **return value**!

```
print("Numbers between 1 and 10: even or odd")

for i in range(1,10):
    if is_even(i):
        print(i, "even")
    else:
        print(i, "odd")
```

Big Idea

Don't write code right away!

Writing Code: Paper First!

Suppose we want to add all the odd integers between (and including) **a** and **b**

Writing Code: Paper First!

Suppose we want to add all the odd integers between (and including) **a** and **b**

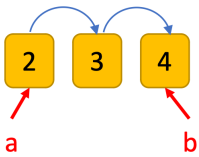
- Start with a **simple example on paper**
- Systematically solve example

```
def sum_odd(a, b):
    # your code here
    return sum_of_odds
```

Writing Code: Paper First!

Suppose we want to add all the odd integers between (and including) **a** and **b**

- Start with a **simple example on paper**
- a = 2 and b = 4
  - `sum_of_odds` should be **3**

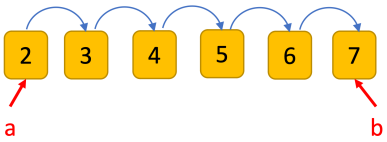



```
def sum_odd(a, b):
    # your code here
    return sum_of_odds
```

Writing Code: Paper First!

Suppose we want to add all the odd integers between (and including) **a** and **b**

- Start with a **simple example on paper**
- a = 2 and b = 7
  - `sum_of_odds` should be **15**




```
def sum_odd(a, b):
    # your code here
    return sum_of_odds
```

Writing Code: Paper First!

- Add **ALL** numbers between *(and including)* a and b
  - It's a Loop

```
def sum_odd(a, b):
    # your code here
    return sum_of_odds
```

Writing Code: Paper First!

- Add **ALL** numbers between *(and including)* a and b
  - It's a Loop
- `while` or `for` ?

```
def sum_odd(a, b):
    # your code here
    return sum_of_odds
```

for loop

```
def sum_odd(a, b):
    for i in range(a,b):
        # do something
    return sum_of_odds
```

while loop

```
def sum_odd(a, b):
    i = a
    while i <= b:
        # do something
        i += 1
    return sum_of_odds
```

for loop

```
def sum_odd(a, b):
    sum_of_odds = 0
    for i in range(a,b):
        sum_of_odds += i
    return sum_of_odds
```

while loop

```
def sum_odd(a, b):
    sum_of_odds = 0
    i = a
    while i <= b:
        sum_of_odds += i
        i += 1
    return sum_of_odds
```

for loop

```
def sum_odd(a, b):
    sum_of_odds = 0
    for i in range(a,b+1):
        if i%2 != 0:
            sum_of_odds += i
    return sum_of_odds
```

while loop

```
def sum_odd(a, b):
    sum_of_odds = 0
    i = a
    while i <= b:
        if i%2 != 0:
            sum_of_odds += i
        i += 1
    return sum_of_odds
```

for loop

```
def sum_odd(a, b):
    sum_of_odds = 0
    for i in range(a,b+1):
        if i%2 != 0:
            sum_of_odds += i
            # always debug
            print(i,sum_of_odds)
    return sum_of_odds
```

while loop

```
def sum_odd(a, b):
    sum_of_odds = 0
    i = a
    while i <= b:
        if i%2 != 0:
            sum_of_odds += i
            # always debug
            print(i,sum_of_odds)
        i += 1
    return sum_of_odds
```

Decomposition: Functions Calling Functions

Rewrite sum_odd using our is_even function:

```
def is_odd(i):
    return not is_even(i)

def sum_odd(a, b):
    total = 0
    for i in range(a, b+1):
        if is_odd(i):
            total += i
    return total
```

Decomposition: Functions Calling Functions

Rewrite sum_odd using our is_even function:

```
def is_odd(i):
    return not is_even(i)

def sum_odd(a, b):
    total = 0
    for i in range(a, b+1):
        if is_odd(i):
            total += i
    return total
```

You Try!

Write code that satisfies the following specs

```
def is_divisible_by_3_and_5(n):
    """ n is a positive integer
    Returns True if n is divisible by both 3 and 5
    Hint: use your div_by function! """
```

Test your code with:

```
>>> is_divisible_by_3_and_5(15)
True
>>> is_divisible_by_3_and_5(9)
False
>>> is_divisible_by_3_and_5(30)
True
```

You Try!

Write code that satisfies the following specs

```
def is_palindrome(s):
    """ s is a string
    Returns True if s is a palindrome
        and False otherwise
    """
```

For example:

```
>>> is_palindrome("madam")
True
>>> is_palindrome("abc")
False
```

Default Parameters

A Simple power Function

We wrote `square(x)` earlier. But what if we want cubes too?

```
def power(x):
    return x ** 2

print(power(5))    # 25
print(power(3))    # 9
```

A Simple power Function

We wrote `square(x)` earlier. But what if we want cubes too?

```
def power(x):
    return x ** 2

print(power(5))    # 25
print(power(3))    # 9
```

This function **always** squares — what if we want `x**3` or `x**4`?

Adding a Parameter

- **Motivation:** want to compute **any** power, not just squares
- **Options:**
  - Change the exponent **inside function** (all calls are affected)
  - Use an exponent **outside function** (global variables are bad)
  - Add the exponent as **an argument** to the function

n as a Parameter

```
def power(x, n):
    return x ** n

print(power(5, 2))      # 25
print(power(5, 3))      # 125
```

n as a Parameter

```
def power(x, n):
    return x ** n

print(power(5, 2))      # 25
print(power(5, 3))      # 125
```

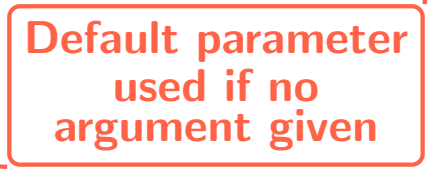
But most of the time we just want to **square** — do we **always** have to pass n?

Default Parameter

```
def power(x, n=2):
    return x ** n

print(power(5))       # 25
print(power(5, 3))    # 125
```

Default Parameter

```
def power(x, n=2):
    return x ** n

print(power(5))      # 25
print(power(5, 3))   # 125
```



Default Parameter

```
def power(x, n=2):
    return x ** n

print(power(5))        # 25
print(power(5, 3))     # 125
```

Use arg, default value not used

Default Parameters: Rules

In the **function definition**:

- Default parameters must go at the end of the parameter list

Default Parameters: Rules

In the **function definition**:

- Default parameters must go at the end of the parameter list

These are **ok for calling a function**:

```
1 power(5)
2 power(5, 3)
3 power(5, n=3)
4 power(x=5, n=3)
5 power(n=3, x=5)
```

Default Parameters: Rules

In the **function definition**:

- Default parameters must go at the end of the parameter list

These are **ok for calling a function**:

```
power(5)
power(5, 3)
power(5, n=3)
power(x=5, n=3)
power(n=3, x=5)
```

These are **not ok for calling a function**:

```
# error, cannot have positional after keyword:
power(n=3, 5)
power(3, 5)           # no error but wrong
```

Common Mistakes with Functions

1. Forgetting `return` — function gives `None`
   - ```
     def f(x):
         x * 2      ← missing return!
     ```
     ```

Common Mistakes with Functions

1. Forgetting `return` — function gives `None`
   - ```
     def f(x):
         x * 2      ← missing return!
     ```
2. Using `print` instead of `return` — can't store or reuse the result
     ```

Common Mistakes with Functions

1. Forgetting `return` — function gives `None`
   - `def f(x):`
     `x * 2` ← missing `return`!
2. Using `print` instead of `return` — can't store or reuse the result
3. Forgetting to **call** the function
   - `is_even` vs `is_even(4)`

Common Mistakes with Functions

1. Forgetting `return` — function gives `None`
   - `def f(x):`
     `x * 2` ← missing `return`!
2. Using `print` instead of `return` — can't store or reuse the result
3. Forgetting to **call** the function
   - `is_even` vs `is_even(4)`
4. Forgetting `b+1` in `range(a, b+1)` when you want to include `b`

Summary

Questions?