61A Lecture 5

Wednesday, September 7
Office Hours: You Should Go!

You are not alone!
Office Hours: You Should Go!

You are not alone!
Office Hours: You Should Go!

You are not alone!

http://inst.eecs.berkeley.edu/~cs61a/fa11/www/staff.html
from operator import floordiv, mod

def divide_exact(n, d):
    """Return the quotient and remainder of dividing n by d."

    >>> q, r = divide_exact(13, 5)
    >>> q
    2
    >>> r
    3
    """

    return floordiv(n, d), mod(n, d)
Reminder: Multiple Assignment & Return Values

from operator import floordiv, mod

def divide_exact(n, d):
    """Return the quotient and remainder of dividing n by d."

    >>> q, r = divide_exact(13, 5)
    >>> q
    2
    >>> r
    3
    """

    return floordiv(n, d), mod(n, d)
Reminder: Multiple Assignment & Return Values

Integer division, which rounds down

```python
from operator import floordiv, mod

def divide_exact(n, d):
    '''Return the quotient and remainder of dividing n by d.''
    return floordiv(n, d), mod(n, d)
```

Integer remainder after dividing

```python
>>> q, r = divide_exact(13, 5)
>>> q
2
>>> r
3
'''
from operator import floordiv, mod

def divide_exact(n, d):
    """Return the quotient and remainder of dividing n by d."

    >>> q, r = divide_exact(13, 5)
    >>> q
    2
    >>> r
    3
    """
    return floordiv(n, d), mod(n, d)

Multiple return values, separated by commas
Reminder: Multiple Assignment & Return Values

```python
from operator import floordiv, mod

def divide_exact(n, d):
    """Return the quotient and remainder of dividing n by d."

    >>> q, r = divide_exact(13, 5)
    >>> q
    2
    >>> r
    3
    """
    return floordiv(n, d), mod(n, d)
```

- Integer division, which rounds down
- Integer remainder after dividing
- Multiple return values, separated by commas
- Multiple assignment to two names
The Structure of Project 1

Two functions implement the game simulation
Two functions implement the game simulation

Warning!
Pseudo-code
(not code)
The Structure of Project 1

Two functions implement the game simulation

```python
def play(...):
    while game is not over:
        plan = get a plan (from the current player's strategy)
        take_turn = call take_turn with a dice and plan
    return winner
```

Warning! Pseudo-code (not code)
The Structure of Project 1

Two functions implement the game simulation

```python
def play(...):
    while game is not over:
        get a plan (from the current player's strategy)
        call take_turn with a dice and plan
    return winner

def take_turn(...):
    while turn is not over:
        get an action (from plan) and outcome (from dice)
        call an action
    return points scored during the turn
```

Warning! Pseudo-code (not code)
### The Structure of Project 1

Four types of functions are involved in simulating game

<table>
<thead>
<tr>
<th>Domain</th>
<th>Range</th>
</tr>
</thead>
</table>

Wednesday, September 7, 2011
The Structure of Project 1

Four types of functions are involved in simulating game

<table>
<thead>
<tr>
<th>Domain</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action</td>
<td>(integer, integer) (integer, integer, boolean)</td>
</tr>
</tbody>
</table>
The Structure of Project 1

Four types of functions are involved in simulating game

<table>
<thead>
<tr>
<th>Action</th>
<th>Domain</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(integer, integer)</td>
<td>(integer, integer, integer, boolean)</td>
</tr>
</tbody>
</table>

Two arguments
The Structure of Project 1

Four types of functions are involved in simulating game

<table>
<thead>
<tr>
<th>Domain</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action</td>
<td>(integer, integer)</td>
</tr>
<tr>
<td></td>
<td>Two arguments</td>
</tr>
<tr>
<td></td>
<td>(integer, integer, boolean)</td>
</tr>
<tr>
<td></td>
<td>Three return values</td>
</tr>
</tbody>
</table>
The Structure of Project 1

Four types of functions are involved in simulating game

<table>
<thead>
<tr>
<th></th>
<th>Domain</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Action</strong></td>
<td>(integer, integer)</td>
<td>(integer, integer, boolean)</td>
</tr>
<tr>
<td><strong>Plan</strong></td>
<td>integer</td>
<td>Action</td>
</tr>
<tr>
<td></td>
<td>Two arguments</td>
<td>Three return values</td>
</tr>
</tbody>
</table>
The Structure of Project 1

Four types of functions are involved in simulating game

<table>
<thead>
<tr>
<th>Domain</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Action</strong></td>
<td>(integer, integer)</td>
</tr>
<tr>
<td><strong>Plan</strong></td>
<td>integer</td>
</tr>
<tr>
<td><strong>Strategy</strong></td>
<td>(integer, integer)</td>
</tr>
</tbody>
</table>
The Structure of Project 1

Four types of functions are involved in simulating game

<table>
<thead>
<tr>
<th>Function</th>
<th>Domain</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Action</strong></td>
<td>(integer, integer)</td>
<td>(integer, integer, boolean)</td>
</tr>
<tr>
<td><strong>Plan</strong></td>
<td>integer</td>
<td>Action</td>
</tr>
<tr>
<td><strong>Strategy</strong></td>
<td>(integer, integer)</td>
<td>Plan</td>
</tr>
<tr>
<td><strong>Dice</strong></td>
<td>No arguments</td>
<td>integer</td>
</tr>
</tbody>
</table>
The Purpose of Higher-Order Functions
The Purpose of Higher-Order Functions

Functions are first-class: Functions can be manipulated as values in our programming language.
The Purpose of Higher-Order Functions

Functions are first-class: Functions can be manipulated as values in our programming language.

Higher-order function: A function that takes a function as an argument value or returns a function as a return value.
The Purpose of Higher-Order Functions

**Functions are first-class:** Functions can be manipulated as values in our programming language.

**Higher-order function:** A function that takes a function as an argument value or returns a function as a return value

Higher-order functions:
The Purpose of Higher-Order Functions

Functions are first-class: Functions can be manipulated as values in our programming language.

Higher-order function: A function that takes a function as an argument value or returns a function as a return value

Higher-order functions:

• Express general methods of computation
The Purpose of Higher-Order Functions

Functions are first-class: Functions can be manipulated as values in our programming language.

Higher-order function: A function that takes a function as an argument value or returns a function as a return value

Higher-order functions:

• Express general methods of computation
• Remove repetition from programs
The Purpose of Higher-Order Functions

Functions are first-class: Functions can be manipulated as values in our programming language.

Higher-order function: A function that takes a function as an argument value or returns a function as a return value.

Higher-order functions:

- Express general methods of computation
- Remove repetition from programs
- Separate concerns among functions
def cube(k):
    return pow(k, 3)

def summation(n, term):
    """Sum the first n terms of a sequence."

    total, k = 0, 1
    while k <= n:
        total, k = total + term(k), k + 1
    return total

>>> summation(5, cube)
225
"""
Review: Summation Example

```python
def cube(k):
    return pow(k, 3)

def summation(n, term):
    """Sum the first n terms of a sequence.
    >>> summation(5, cube)
    225
    """
    total, k = 0, 1
    while k <= n:
        total, k = total + term(k), k + 1
    return total
```

Function of a single argument (not called term)
def cube(k):
    return pow(k, 3)

def summation(n, term):
    """Sum the first n terms of a sequence."

    total, k = 0, 1
    while k <= n:
        total, k = total + term(k), k + 1
    return total

>>> summation(5, cube)
225
"""
def cube(k):
    return pow(k, 3)

def summation(n, term):
    """Sum the first n terms of a sequence."

    total, k = 0, 1
    while k <= n:
        total, k = total + term(k), k + 1
    return total

>>> summation(5, cube)
225
"""
Review: Summation Example

```python
def cube(k):
    return pow(k, 3)

def summation(n, term):
    """Sum the first n terms of a sequence."""
    total, k = 0, 1
    while k <= n:
        total, k = total + term(k), k + 1
    return total

>>> summation(5, cube)
225
"""
```

Function of a single argument (not called term)

A formal parameter that will be bound to a function

The cube function is passed as an argument value

The function bound to term gets called here
def cube(k):
    return pow(k, 3)

def summation(n, term):
    """Sum the first n terms of a sequence."
    total, k = 0, 1
    while k <= n:
        total, k = total + term(k), k + 1
    return total

>>> summation(5, cube)

225

"""0 + 1^3 + 2^3 + 3^3 + 4^3 + 5^5"

Function of a single argument (not called term)

A formal parameter that will be bound to a function

The cube function is passed as an argument value

The function bound to term gets called here
Environments Enable Higher-Order Functions!

Functions as arguments:

Functions as return values:
Environments Enable Higher-Order Functions!

Functions as arguments:

Our current environment model handles that!

Functions as return values:
Environments Enable Higher-Order Functions!

**Functions as arguments:**

Our current environment model handles that!

We'll give an example of how

**Functions as return values:**
Environments Enable Higher-Order Functions!

Functions as arguments:

Our current environment model handles that!

We'll give an example of how

Functions as return values:

We need to extend the model a little
Environments Enable Higher-Order Functions!

Functions as arguments:

Our current environment model handles that!

We'll give an example of how

Functions as return values:

We need to extend the model a little

Functions need to know where they were defined
Environments Enable Higher-Order Functions!

Functions as arguments:

Our current environment model handles that!

We'll give an example of how

Functions as return values:

We need to extend the model a little

Functions need to know where they were defined

Almost everything stays the same
Names and Environments with Functional Values

```python
def apply_twice(f, x):
    return f(f(x))

def square(x):
    return x * x

apply_twice(square, 2)
```
Names and Environments with Functional Values

```python
def apply_twice(f, x):
    return f(f(x))

def square(x):
    return x * x

apply_twice(square, 2)
```
def apply_twice(f, x):
    return f(f(x))

def square(x):
    return x * x

apply_twice(square, 2)
def apply_twice(f, x):
    return f(f(x))

def square(x):
    return x * x

apply_twice(square, 2)
def apply_twice(f, x):
    return f(f(x))

def square(x):
    return x * x

apply_twice(square, 2)
Names and Environments with Functional Values

```python
def apply_twice(f, x):
    return f(f(x))

def square(x):
    return x * x

def apply_twice(square, 2):
    return apply_twice(square, 2)
```

Wednesday, September 7, 2011
def apply_twice(f, x):
    return f(f(x))

def square(x):
    return x * x

apply_twice(square, 2)
```python
def apply_twice(f, x):
    return f(f(x))

def square(x):
    return x * x

apply_twice(square, 2)
```
def apply_twice(f, x):
    return f(f(x))

def square(x):
    return x * x

apply_twice(square, 2)
def apply_twice(f, x):
    return f(f(x))

def square(x):
    return x * x

apply_twice(square, 2)
def apply_twice(f, x):
    return f(f(x))

def square(x):
    return x * x

apply_twice(square, 2)
def apply_twice(f, x):
    return f(f(x))

def square(x):
    return x * x

apply_twice(square, 2)
Applying User-Defined Functions

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

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

```python
def square(x):
    return x * x
square(-2)
```
Applying User-Defined Functions

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

square(-2)
```

---

Wednesday, September 7, 2011
Applying User-Defined Functions

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

square(-2)
```

The first frame of the environment in which the function was defined.
Applying User-Defined Functions

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

The first frame of the environment in which the function was defined.

def square(x):
    return x * x

square(-2)
Applying User-Defined Functions

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

```
square(-2)
```

The first frame of the environment in which the function was defined

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

```
square(-2)
```
Applying User-Defined Functions

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

```
x: -2
```

```
square(-2)
```

The first frame of the environment in which the function was defined
Applying User-Defined Functions

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

```
square(-2)
```

The first frame of the environment in which the function was defined...
Applying User-Defined Functions

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

The first frame of the environment in which the function was defined.
Applying User-Defined Functions

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

square(-2)
```

The first frame of the environment in which the function was defined.
Applying User-Defined Functions

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

The first frame of the environment in which the function was defined
Applying User-Defined Functions

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

The first frame of the environment in which the function was defined.
Functions Associated with the Global Frame

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

square(-2)
```

Functions Associated with the Global Frame

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

```
x: -2
```

```
square(-2): return x * x
```

Associated with the global frame
Functions Associated with the Global Frame

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

square(-2)
```

This is the global frame

Associated with the global frame

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

square(-2)
```

Wednesday, September 7, 2011
Locally Defined Functions: Example
Locally Defined Functions: Example

Functions defined within other function bodies are bound to names in the local frame
Locally Defined Functions: Example

Functions defined within other function bodies are bound to names in the local frame

```python
def make_adder(n):
    """Return a function that takes one argument k and returns k + n."""

    def adder(k):
        return k + n
    return adder

>>> add_three = make_adder(3)
>>> add_three(4)
7
"""
```

Functions defined within other function bodies are bound to names in the local frame.
Locally Defined Functions: Example

Functions defined within other function bodies are bound to names in the local frame

A function that returns a function

def make_adder(n):
    """Return a function that takes one argument k and returns k + n."
    
    def adder(k):
        return k + n
    return adder

>>> add_three = make_adder(3)
>>> add_three(4)
7

# Local function definitions; returning functions

def make_adder(n):
    """Return a function that takes one argument k and returns k + n."
    
    def adder(k):
        return k + n
    return adder

def compose1(f, g):
    """Return a function that composes f and g."
    
    def h(x):
        return f(g(x))
    return h

@main
def run():
    interact()
Locally Defined Functions: Example

Functions defined within other function bodies are bound to names in the local frame.

A function that returns a function

```python
def make_adder(n):
    """Return a function that takes one argument k and returns k + n."""
    def adder(k):
        return k + n
    return adder

>>> add_three = make_adder(3)
>>> add_three(4)
7

The name add_three is bound to a function.
Locally Defined Functions: Example

Functions defined within other function bodies are bound to names in the local frame.

A function that returns a function

```python
def make_adder(n):
    """Return a function that takes one argument k and returns k + n."
    return lambda k: k + n
```

The name add_three is bound to a function

```python
def add_three(k):
    return k + 3
```

A local def statement
Locally Defined Functions: Example

Functions defined within other function bodies are bound to names in the local frame

A function that returns a function

```python
def make_adder(n):
    return lambda k: k + n
```

The name add_three is bound to a function

```python
```

A local def statement

```python
def adder(k):
    return k + n
```

Can refer to names in the enclosing function

```python
add_three = make_adder(3)
add_three(4)
```
Locally Defined Functions: Call Expressions

```
def make_adder(n):
    def adder(k):
        return k + n
    return adder

make_adder(1)(2)
```
Locally Defined Functions: Call Expressions

make_adder(1)(2)

make_adder(1) ( 2 )

def make_adder(n):
    def adder(k):
        return k + n
    return adder

make_adder(1)(2)
Locally Defined Functions: Call Expressions

\[
\text{make\_adder}(1)(2)
\]

\[
\text{Operator}
\]

```python
def make_adder(n):
    def adder(k):
        return k + n
    return adder
make_adder(1)(2)
```
Locally Defined Functions: Call Expressions

make_adder(1)(2)

```
def make_adder(n):
    def adder(k):
        return k + n
    return adder

make_adder(1)(2)
```
Locally Defined Functions: Call Expressions

make_adder(1)(2)

```
def make_adder(n):
    def adder(k):
        return k + n
    return adder

def make_adder(1)(2)
```

An expression that evaluates to a function
Locally Defined Functions: Call Expressions

make_adder(1)(2)

def make_adder(n):
    def adder(k):
        return k + n
    return adder

make_adder(1)(2)
Locally Defined Functions: Environments

```python
def make_adder(n):
    def adder(k):
        return k + n
    return adder

make_adder(1)(2)
```
Locally Defined Functions: Environments

```
def make_adder(n):
    def adder(k):
        return k + n
    return adder

make_adder(1)(2)
```
Locally Defined Functions: Environments

```
def make_adder(n):
    def adder(k):
        return k + n
    return adder

make_adder(1)(2)
```
Locally Defined Functions: Environments

```
def make_adder(n):
    def adder(k):
        return k + n
    return adder

make_adder(1)(2)
```

**Diagram Explanation:**
- **`make_adder`**
  - Arguments: `n`
  - Returns: `adder`
- **`adder`**
  - Argument: `k`
  - Returns: `k + n`
- **`make_adder(1)(2)`**
  - Evaluates to `3`
Locally Defined Functions: Environments

```python
def make_adder(n):
    def adder(k):
        return k + n
    return adder

make_adder(1)(2)
```

Associated with a local frame
Locally Defined Functions: Environments

make_adder:

n: 1

adder:

Associated with a local frame

def make_adder(n):
    def adder(k):
        return k + n
    return adder

make_adder(1)(2)

make_adder(1)
Locally Defined Functions: Environments

```python
def make_adder(n):
    def adder(k):
        return k + n
    return adder

make_adder(1)(2)
```

Associated with a local frame

Apply `adder` to 2
Locally Defined Functions: Environments

```
def make_adder(n):
    def adder(k):
        return k + n
    return adder

make_adder(1)(2)
```

```
def make_adder(n):
    def adder(k):
        return k + n
    return adder

make_adder
```

```
made_adder(1)
made_adder(n):
... make_adder(n):

time
time

make_adder
```

```
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
adder:
Locally Defined Functions: Environments

```
def make_adder(n):
    def adder(k):
        return k + n
    return adder

make_adder(1)(2)
```

Associated with a local frame

Apply `adder` to 2

`make_adder(1)(2)`

`return k + n`
def make_adder(n):
    def adder(k):
        return k + n
    return adder

make_adder(1)(2)
Locally Defined Functions: Environments

```
def make_adder(n):
    def adder(k):
        return k + n
    return adder

make_adder(1)(2)
```

Associated with a local frame

Apply `adder` to 2

```
def make_adder(n):
    def adder(k):
        return k + n
    return adder

make_adder(1)(2)
```

```
def make_adder(n):
    def adder(k):
        return k + n
    return adder

make_adder(1)
```
The Environment for Function Composition

```python
def compose1(f, g):
    def h(x):
        return f(g(x))
    return h
a1 = make_adder(1)
a2 = make_adder(2)
compose1(a1, a2)(3)
```
(Demo)
The Environment for Function Composition

```python
def compose1(f, g):
    def h(x):
        return f(g(x))
    return h

a1 = make_adder(1)
a2 = make_adder(2)
compose1(a1, a2)(3)
```

(Demo)
The Environment for Function Composition

```python
def compose1(f, g):
    def h(x):
        return f(g(x))
    return h

da1 = make_adder(1)
da2 = make_adder(2)
compose1(a1, a2)(3)
```

(Demo)
The Environment for Function Composition

```python
make_adder:
make_adder(n):

adder:
adder(k):
return k + n

compose1:
def compose1(f, g):
    def h(x):
        return f(g(x))
    return h

def make_adder(n):
    ...

a1 = make_adder(1)
a2 = make_adder(2)
compose1(a1, a2)(3)
```

(Demo)
The Environment for Function Composition

```python
def compose1(f, g):
    def h(x):
        return f(g(x))
    return h

a1 = make_adder(1)
a2 = make_adder(2)
compose1(a1, a2)(3)
```

(Demo)
The Environment for Function Composition

```python
def make_adder(n):
    def adder(k):
        return k + n
    return adder

a1 = make_adder(1)
a2 = make_adder(2)
compose1(a1, a2)(3)
```

(Demo)
The Environment for Function Composition

```
make_adder:
make_adder(n):
    return k + n

adder:
adder(k):
    return k + n

compose1:
def compose1(f, g):
    def h(x):
        return f(g(x))
    return h

da1 = make_adder(1)
da2 = make_adder(2)
compose1(a1, a2)(3)
```
The Environment for Function Composition

```
def adder(k):
    return k + n

complement:
def make_adder(n):
    return adder

compose1: def compose1(f, g):
    def h(x):
        return f(g(x))
    return h

a1 = make_adder(1)
a2 = make_adder(2)
compose1(a1, a2)(3)
```

(Demo)
The Environment for Function Composition

```python
def make_adder(n):
    return n + n

def compose1(f, g):
    def h(x):
        return f(g(x))
    return h

a1 = make_adder(1)
a2 = make_adder(2)
compose1(a1, a2)(3)
```

(Demo)
The Environment for Function Composition

```
def make_adder(n):
    return make_adder(n):

def compose1(f, g):
    def h(x):
        return f(g(x))
    return h

a1 = make_adder(1)
a2 = make_adder(2)
compose1(a1, a2)(3)
```

(Demo)
The Environment for Function Composition

```python
def compose1(f, g):
    def h(x):
        return f(g(x))
    return h

a1 = make_adder(1)
a2 = make_adder(2)
compose1(a1, a2)(3)
```

(Demo)
Lambda Expressions
Lambda Expressions

```python
>>> ten = 10
```
Lambda Expressions

```python
>>> ten = 10

>>> square = x * x
```
Lambda Expressions

>>> ten = 10

>>> square = x * x

An expression: this one evaluates to a number
Lambda Expressions

>>> ten = 10

>>> square = \(x \times x\)

>>> square = lambda x: x * x

An expression: this one evaluates to a number
Lambda Expressions

```python
>>> ten = 10

>>> square = x * x

>>> square = lambda x: x * x
```

An expression: this one evaluates to a number

Also an expression: evaluates to a function
Lambda Expressions

```python
>>> ten = 10
An expression: this one evaluates to a number

>>> square = x * x
Also an expression: evaluates to a function

>>> square = lambda x: x * x
A function
```
Lambda Expressions

```python
>>> ten = 10

>>> square = x * x

>>> square = lambda x: x * x
```

An expression: this one evaluates to a number

Also an expression: evaluates to a function

A function with formal parameter x
Lambda Expressions

>>> ten = 10

An expression: this one evaluates to a number

>>> square = \(x \times x\)

Also an expression: evaluates to a function

>>> square = lambda x: x * x

A function

with formal parameter \(x\)

and body "return \(x \times x\)"
Lambda Expressions

```python
>>> ten = 10
An expression: this one evaluates to a number

>>> square = x * x
Also an expression: evaluates to a function

>>> square = lambda x: x * x
A function with formal parameter x
and body "return x * x"
```

Notice: no "return"
Lambda Expressions

```python
>>> ten = 10

>>> square = x * x

>>> square = lambda x: x * x

An expression: this one evaluates to a number

Also an expression: evaluates to a function

A function

with formal parameter x

and body "return x * x"

Notice: no "return"

Must be a single expression
```
Lambda Expressions

>>> ten = 10
An expression: this one evaluates to a number

>>> square = x * x
Also an expression: evaluates to a function

A function
with formal parameter x
and body "return x * x"

>>> square = lambda x: x * x
Notice: no "return"

>>> square(4)
16
Must be a single expression
Lambda Expressions

>>> ten = 10
An expression: this one evaluates to a number

>>> square = \( x \times x \)
Also an expression: evaluates to a function

>>> square = \( \text{lambda } x: x \times x \)
A function
with formal parameter \( x \)
and body "return \( x \times x \)"

>>> square(4)
16
Must be a single expression

Lambda expressions are rare in Python, but important in general
More Higher-Order Function Examples

(Demo)