Locally Defined Functions

Example: [http://goo.gl/pnU8f](http://goo.gl/pnU8f)
Locally Defined Functions

The inner definition is executed each time the outer function is called

Example: http://goo.gl/pnU8f
The inner definition is executed each time the outer function is called.

Example: [http://goo.gl/pnU8f](http://goo.gl/pnU8f)
def make_adder(n):
    """Return a function that adds n to its argument."

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

def adder(k):
    return add(n, k)

return adder
Locally defined functions can be returned

```python
def make_adder(n):
    """Return a function that adds n to its argument."

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

def adder(k):
    return add(n, k)

return adder
```
Functions as Return Values

Locally defined functions can be returned
They have access to the frame in which they are defined

```python
def make_adder(n):
    """Return a function that adds n to its argument."

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

def adder(k):
    return add(n, k)
return adder
```
Locally defined functions can be returned
They have access to the frame in which they are defined

A function that returns a function

```python
def make_adder(n):
    """Return a function that adds n to its argument."
    return add(n, k)

def adder(k):
    return add(n, k)

>>> add_three = make_adder(3)
>>> add_three(4)
7
"""
```
Locally defined functions can be returned
They have access to the frame in which they are defined

```
def make_adder(n):
    """Return a function that adds n to its argument."
    add = adder
    return add

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

```
def adder(k):
    return add(n, k)
return adder
```
Locally defined functions can be returned
They have access to the frame in which they are defined

```python
def make_adder(n):
    """Return a function that adds n to its argument."""
    return add(n,)

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

```
Functions as Return Values

Locally defined functions can be returned
They have access to the frame in which they are defined

```python
def make_adder(n):
    """Return a function that adds n to its argument."
    return lambda k: add(n, k)

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

def adder(k):
    return add(n, k)
return adder
```

A function that returns a function

The name add_three is bound to a function

A local def statement

Can refer to names in the enclosing function
Higher-Order Functions

Functions are first-class: they can be manipulated as values in Python
Functions are first-class: they can be manipulated as values in Python.

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

Functions are first-class: they can be manipulated as values in Python

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

Higher order functions:
 Functions are first-class: they can be manipulated as values in Python

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
Functions are first-class: they can be manipulated as values in Python

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
Higher-Order Functions

Functions are first-class: they can be manipulated as values in Python

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
Environment of Function Application
The environment in which a function is applied consists of:
Environment of Function Application

The environment in which a function is applied consists of:

- A new local frame each time the function is applied
The environment in which a function is applied consists of:

- A new local frame *each* time the function is applied
- The environment in which the function was *defined*
The environment in which a function is applied consists of:

- A new local frame *each* time the function is applied
- The environment in which the function was *defined*
  - Before, this was just the global frame
Environment of Function Application

The environment in which a function is applied consists of:

- A *new* local frame *each* time the function is *applied*

- The environment in which the function was *defined*
  - Before, this was just the global frame
  - For a locally-defined function, this includes all local frames in the definition environment, plus the global frame
The environment of a function application is a new local frame plus the environment in which the function was defined

Example: http://goo.gl/73anC
The environment of a function application is a new local frame plus the environment in which the function was defined.

Example: [http://goo.gl/73anC](http://goo.gl/73anC)
The environment of a function application is a new local frame plus the environment in which the function was defined.

Example: [http://goo.gl/73anC](http://goo.gl/73anC)
Environment for Non-Nested Function

The environment of a function application is a new local frame plus the environment in which the function was defined.

Example: [http://goo.gl/73anC](http://goo.gl/73anC)
Environment for Non-Nested Function

The environment of a function application is a new local frame plus the environment in which the function was defined.

Example: [http://goo.gl/73anC](http://goo.gl/73anC)
The environment of a function application is a new local frame plus the environment in which the function was defined.

Example: [http://goo.gl/73anC](http://goo.gl/73anC)
The environment of a function application is a new local frame plus the environment in which the function was defined.

Example: [http://goo.gl/73anC](http://goo.gl/73anC)
The environment of a function application is a new local frame plus the environment in which the function was defined.

Example: [http://goo.gl/73anC](http://goo.gl/73anC)
Environment for Nested Function

```python
1 def make_adder(n):
2     def adder(k):
3         return k + n
4     return adder

6 add_three = make_adder(3)
7 result = add_three(4)
```
Environment for Nested Function

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

add_three = make_adder(3)
result = add_three(4)
```
Environment for Nested Function

```
1. def make_adder(n):
   2.     def adder(k):
   3.         return k + n
   4.     return adder
   5. 6. add_three = make_adder(3)
   7. result = add_three(4)
```

Diagram:
- Global frame
  - make_adder
  - adder
  - add_three
- f1: make_adder
  - n: 3
  - adder
  - Return value
- adder [parent=f1]
  - k: 4
  - Return value: 7

Nested def
Environment for Nested Function

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

add_three = make_adder(3)
result = add_three(4)
```
Environment for Nested Function

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

add_three = make_adder(3)
result = add_three(4)
```
Environment for Nested Function

```
1 def make_adder(n):
2     def adder(k):
3         return k + n
4         return adder
5
6 add_three = make_adder(3)
7 result = add_three(4)
```
Environment for Nested Function

```
1: def make_adder(n):
2:     def adder(k):
3:         return k + n
4:     return adder
5: 6: add_three = make_adder(3)
7: result = add_three(4)
```
Environment for Nested Function

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

add_three = make_adder(3)
result = add_three(4)
```
Every user-defined function has a parent frame
Every user-defined function has a parent frame.

The parent frame of a function is the frame in which it was defined.
Every user-defined function has a parent frame.
The parent frame of a function is the frame in which it was defined.
Every local frame has a parent frame.
Every user-defined function has a parent frame.

The parent frame of a function is the frame in which it was defined.

Every local frame has a parent frame.

The parent of a local frame is the parent of the function called.
Every user-defined function has a parent frame
The parent frame of a function is the frame in which it was **defined**
Every local frame has a parent frame
The parent of a local frame is the parent of the function **called**
Every user-defined function has a parent frame.
The parent frame of a function is the frame in which it was defined.
Every local frame has a parent frame.
The parent of a local frame is the parent of the function called.
Every user-defined function has a parent frame.
The parent frame of a function is the frame in which it was defined.
Every local frame has a parent frame.
The parent of a local frame is the parent of the function called.
The Structure of Environments

A frame extends the environment that begins with its parent.
The Structure of Environments

A frame extends the environment that begins with its parent.
The Structure of Environments

A frame extends the environment that begins with its parent.
The Structure of Environments

A frame extends the environment that begins with its parent.
The Structure of Environments

A frame extends the environment that begins with its parent

When a frame or function has no label [parent=___] then its parent is always the global frame.
The Structure of Environments

A frame extends the environment that begins with its parent

A three-frame environment

When a frame or function has no label

[parent=___]

then its parent is always the global frame
The Structure of Environments

A frame extends the environment that begins with its parent

When a frame or function has no label

\[\text{[parent=___]}\]

then its parent is always the global frame

A three-frame environment

A two-frame environment
A frame extends the environment that begins with its parent.

The global environment: the environment with only the global frame.

When a frame or function has no label
  \([\text{parent}=\_\_\_]\)
then its parent is always the global frame.

A three-frame environment
A two-frame environment
A three-frame environment
The Structure of Environments

A frame extends the environment that begins with its parent

The global environment: the environment with only the global frame

When a frame or function has no label

\[\text{[parent=___]}\]

then its parent is always the global frame
The Structure of Environments

A frame extends the environment that begins with its parent.

- **Global frame**: the environment with only the global frame.
- **A two-frame environment**: always extends.
- **A three-frame environment**: always extends.

When a frame or function has no label:

\[ \text{[parent=___]} \]

then its parent is always the global frame.
How to Draw an Environment Diagram
How to Draw an Environment Diagram

When defining a function:
How to Draw an Environment Diagram

When defining a function:

Create a function value with signature
\(<\text{name}\>(<\text{formal parameters}>\))
How to Draw an Environment Diagram

When defining a function:

Create a function value with signature
<name>(<formal parameters>)

For nested definitions, label the parent as the first frame of the current environment
How to Draw an Environment Diagram

When defining a function:

Create a function value with signature 
<name>(<formal parameters>)

For nested definitions, label the parent as the first frame of the current environment

Bind <name> to the function value in the first frame of the current environment
How to Draw an Environment Diagram

When defining a function:

Create a function value with signature
<name>(<formal parameters>)

For nested definitions, label the parent as the first frame of the current environment

Bind <name> to the function value in the first frame of the current environment

When calling a function:
How to Draw an Environment Diagram

When defining a function:

Create a function value with signature
\texttt{\textless name\textgreater (\textless formal parameters\textgreater )}

For nested definitions, label the parent as the first frame of the current environment

Bind \texttt{name} to the function value in the first frame of the current environment

When calling a function:

1. Add a local frame labeled with the \texttt{name} of the function
How to Draw an Environment Diagram

When defining a function:

Create a function value with signature

\texttt{<name>(<formal\ parameters>)}

For nested definitions, label the parent as the first frame of the current environment

Bind \texttt{<name>} to the function value in the first frame of the current environment

When calling a function:

1. Add a local frame labeled with the \texttt{<name>} of the function
2. If the function has a parent label, copy it to this frame
How to Draw an Environment Diagram

When defining a function:

Create a function value with signature
<name>(<formal parameters>)

For nested definitions, label the parent as the first frame of the current environment

Bind <name> to the function value in the first frame of the current environment

When calling a function:

1. Add a local frame labeled with the <name> of the function
2. If the function has a parent label, copy it to this frame
3. Bind the <formal parameters> to the arguments in this frame
How to Draw an Environment Diagram

When defining a function:

Create a function value with signature
<name>(<formal parameters>)

For nested definitions, label the parent as the first frame of the current environment

Bind <name> to the function value in the first frame of the current environment

When calling a function:

1. Add a local frame labeled with the <name> of the function
2. If the function has a parent label, copy it to this frame
3. Bind the <formal parameters> to the arguments in this frame
4. Execute the body of the function in the environment that starts with this frame
Environment for Function Composition

Example: http://goo.gl/5zcug
Environment for Function Composition

```python
1 def square(x):
2     return x * x
3
4 def make_adder(n):
5     def adder(k):
6         return n + k
7     return adder
8
9 def compose1(f, g):
10    def h(x):
11        return f(g(x))
12    return h
13
14 compose1(square, make_adder(2))(3)
```

Example: [http://goo.gl/5zcug](http://goo.gl/5zcug)
Environment for Function Composition

Example: http://goo.gl/5zcug
Environment for Function Composition

Example: [http://goo.gl/5zcug](http://goo.gl/5zcug)
Environment for Function Composition

Example: [http://goo.gl/5zcug](http://goo.gl/5zcug)
Environment for Function Composition

Example:

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

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

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

compose1(square, make_adder(2))(3)
```

Return value of `make_adder` is an argument to `compose1`

Example: [http://goo.gl/5zcug](http://goo.gl/5zcug)
Environment for Function Composition

Example:

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

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

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

compose1(square, make_adder(2))(3)
```

Return value of `make_adder` is an argument to `compose1`

Example: [http://goo.gl/5zcug](http://goo.gl/5zcug)
Environment for Function Composition

```
1 def square(x):
2     return x * x
3 def make_adder(n):
4     def adder(k):
5         return n + k
6     return adder
7 def compose1(f, g):
8     def h(x):
9         return f(g(x))
10    return h
11 compose1(square, make_adder(2))(3)
```

Return value of `make_adder` is an argument to `compose1`

Example: [http://goo.gl/5zcug](http://goo.gl/5zcug)
Environment for Function Composition

Example:

```python
1. def square(x):
   return x * x
2. def make_adder(n):
   def adder(k):
       return n + k
   return adder
3. def compose1(f, g):
   def h(x):
       return f(g(x))
   return h
4. compose1(square, make_adder(2))(3)
```

Return value of make_adder is an argument to compose1

Example: [http://goo.gl/5zcug](http://goo.gl/5zcug)
Environment for Function Composition

Example:

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

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

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

compose1(square, make_adder(2))(3)
```

Return value of `make_adder` is an argument to `compose1`
Environment for Function Composition

Example: http://goo.gl/5zcug
Environment for Function Composition

Example: [http://goo.gl/5zcug](http://goo.gl/5zcug)
Environment for Function Composition

Example: [http://goo.gl/5zcug](http://goo.gl/5zcug)
Lambda Expressions

>>> ten = 10
>>> ten = 10

>>> square = x * x
Lambda Expressions

```python
>>> ten = 10

>>> square = 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
Lambda Expressions

>>> ten = 10

>>> square = x * x

An expression: this one evaluates to a number

>>> square = lambda x: x * x

Also an expression: evaluates to a function
Lambda Expressions

>>> 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
Lambda Expressions

```python
>>> ten = 10
```

```python
>>> square = x * x
```

An expression: this one evaluates to a number

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

Also an expression: evaluates to a function

A function with formal parameter x
Lambda Expressions

```python
>>> ten = 10
```

```python
>>> square = x * x
```

```python
>>> 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 \(x \times x\)"
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
Notice: no "return"
A function with formal parameter x and body "return x * x"
```
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

Notice: no "return"

A function

with formal parameter `x`

and body "return `x * x`"

Must be a single expression
Lambda Expressions

```python
>>> ten = 10

>>> square = x * x

>>> square = lambda x: x * x

>>> square(4)
16
```

An expression: this one evaluates to a number

Also an expression: evaluates to a function

Notice: no "return"

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

Must be a single expression
Lambda Expressions

```
>>> ten = 10

>>> square = x * x

>>> square = lambda x: x * x

>>> square(4)
16
```

Lambda expressions are rare in Python, but important in general.
Evaluation of Lambda vs. Def
Evaluation of Lambda vs. Def

\[ \text{lambda } x: x \times x \]
Evaluation of Lambda vs. Def

\[ \text{lambda } x: x \ast x \text{ VS } \]
Evaluation of Lambda vs. Def

\[ \text{lambda } x: x \ast x \quad \text{VS} \quad \text{def } \text{square}(x): \text{return } x \ast x \]
Evaluation of Lambda vs. Def

\[
\text{lambda } x: x \times x \quad \text{ VS } \quad \text{def } \text{square}(x): \text{return } x \times x
\]

Execution procedure for def statements:
Evaluation of Lambda vs. Def

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

Execution procedure for def statements:
1. Create a function value with signature `<name>(<formal parameters>)` and the current frame as parent
Evaluation of Lambda vs. Def

\texttt{lambda \textit{x}: \textit{x} * \textit{x}} VS \texttt{def square(x): return \textit{x} * \textit{x}}

Execution procedure for def \textit{statements}:
1. Create a function value with signature \texttt{<name>(<formal parameters>)} and the current frame as parent
2. Bind \texttt{<name>} to that value in the current frame
Evaluation of Lambda vs. Def

\[ \text{lambda } x: x \times x \quad \text{VS} \quad \text{def } \text{square}(x): \text{return } x \times x \]

Execution procedure for def \textit{statements}:
1. Create a function value with signature \texttt{<name>(<formal parameters>)} and the current frame as parent
2. Bind \texttt{<name>} to that value in the current frame

Evaluation procedure for lambda \textit{expressions}:
Evaluation of Lambda vs. Def

\[ \text{lambda } x: x \times x \quad \text{VS} \quad \text{def} \text{ square}(x): \text{return} \ x \times x \]

Execution procedure for def \textit{statements}:
1. Create a function value with signature \(<\text{name}>(<\text{formal parameters}>)\) and the current frame as parent
2. Bind \(<\text{name}>\) to that value in the current frame

Evaluation procedure for lambda \textit{expressions}:
1. Create a function value with signature \(\lambda(<\text{formal parameters}>)\) and the current frame as parent
Evaluation of Lambda vs. Def

\[
\text{lambda x: x * x} \quad \text{VS} \quad \text{def square(x): return x * x}
\]

Execution procedure for def statements:
1. Create a function value with signature \(<\text{name}>(<\text{formal parameters}>))\) and the current frame as parent
2. Bind \(<\text{name}>\) to that value in the current frame

Evaluation procedure for lambda expressions:
1. Create a function value with signature \(\lambda(<\text{formal parameters}>))\) and the current frame as parent

No intrinsic name
Evaluation of Lambda vs. Def

\[
\text{\texttt{lambda } x: x \times x} \quad \text{VS} \quad \text{\texttt{def } square(x): return x \times x}
\]

Execution procedure for def statements:
1. Create a function value with signature \(<\text{name}>(<\text{formal parameters}>)\) and the current frame as parent
2. Bind \(<\text{name}>\) to that value in the current frame

Evaluation procedure for lambda expressions:
1. Create a function value with signature \(\lambda (<\text{formal parameters}>)\) and the current frame as parent
2. Evaluate to that value

No intrinsic name
Lambda vs. Def Statements
Lambda vs. Def Statements

\[
square = \lambda x: x * x
\]
Lambda vs. Def Statements

\[
\text{square} = \lambda \text{x}: \ x \times x \quad \text{VS}
\]
Lambda vs. Def Statements

\[
\text{square} = \text{lambda } x: x \times x \quad \text{VS} \quad \text{def } \text{square}(x): \\
\text{return } x \times x
\]
Lambda vs. Def Statements

\[ \text{square} = \lambda x : x \ast x \quad \text{VS} \quad \text{def} \ \text{square}(x) : \\
\text{return} \ x \ast x \]

Both create a function with the same arguments & behavior
Lambda vs. Def Statements

\[
square = \text{lambda } x: x * x \quad \text{VS} \quad \text{def } \text{square}(x) :
\]
\[
    \text{return } x * x
\]

Both create a function with the same arguments & behavior

Both of those functions are associated with the environment in which they are defined
Lambda vs. Def Statements

\[
square = \text{lambda } x: x * x \quad \text{VS} \quad \text{def square(x): return } x * x
\]

Both create a function with the same arguments & behavior

Both of those functions are associated with the environment in which they are defined

Both bind that function to the name "square"
Lambda vs. Def Statements

\[
\text{square} = \text{lambda } x: x \times x \quad \text{VS} \quad \text{def } \text{square}(x): \\
\text{return } x \times x
\]

Both create a function with the same arguments & behavior

Both of those functions are associated with the environment in which they are defined

Both bind that function to the name "square"

Only the def statement gives the function an intrinsic name
Lambda vs. Def Statements

\[ \text{square} = \lambda x: x \times x \quad \text{VS} \quad \text{def} \ \text{square}(x): \]
\[ \quad \text{return} \ x \times x \]

Both create a function with the same arguments & behavior

Both of those functions are associated with the environment in which they are defined

Both bind that function to the name "square"

Only the def statement gives the function an intrinsic name
Lambda vs. Def Statements

\[ \text{square} = \text{lambda } x: x \times x \quad \text{VS} \quad \text{def square}(x): \text{return } x \times x \]

Both create a function with the same arguments & behavior

Both of those functions are associated with the environment in which they are defined

Both bind that function to the name "square"

Only the def statement gives the function an intrinsic name
Lambda vs. Def Statements

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

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

Both create a function with the same arguments & behavior

Both of those functions are associated with the environment in which they are defined

Both bind that function to the name "square"

Only the def statement gives the function an intrinsic name
Lambda vs. Def Statements

square = lambda x: x * x  

VS

def square(x):
    return x * x

Both create a function with the same arguments & behavior

Both of those functions are associated with the environment in which they are defined

Both bind that function to the name "square"

Only the def statement gives the function an intrinsic name