CS61A Lecture 4

Amir Kamil
UC Berkeley
January 30, 2013
Announcements

- Reminder: hw1 due tonight

- In-class quiz on Friday
  - Covers through Wednesday’s lecture
  - Bring a writing implement

- Hog project out
  - Get started early!
  - Try it out online! See the announcement on the website
The Game of Hog
The Game of Hog

Expected Score

<table>
<thead>
<tr>
<th>Number of Dice Rolled (6-Sided Dice)</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.7</td>
</tr>
<tr>
<td>2</td>
<td>6.0</td>
</tr>
<tr>
<td>3</td>
<td>7.5</td>
</tr>
<tr>
<td>4</td>
<td>8.4</td>
</tr>
<tr>
<td>5</td>
<td>8.9</td>
</tr>
<tr>
<td>6</td>
<td>8.9</td>
</tr>
<tr>
<td>7</td>
<td>8.7</td>
</tr>
<tr>
<td>8</td>
<td>8.5</td>
</tr>
<tr>
<td>9</td>
<td>8.0</td>
</tr>
<tr>
<td>10</td>
<td>7.5</td>
</tr>
</tbody>
</table>
The Game of Hog

Expected Score

<table>
<thead>
<tr>
<th>Number of Dice Rolled (6-Sided Dice)</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3.7</td>
<td>6.0</td>
<td>7.5</td>
<td>8.4</td>
<td>8.9</td>
<td>8.9</td>
<td>8.7</td>
<td>8.5</td>
<td>8.0</td>
<td>7.5</td>
</tr>
</tbody>
</table>

Chance of 10+

<table>
<thead>
<tr>
<th>Number of Dice Rolled (6-Sided Dice)</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>17%</td>
<td>49%</td>
<td>48%</td>
<td>40%</td>
<td>33%</td>
<td>28%</td>
<td>23%</td>
<td>19%</td>
<td>16%</td>
<td></td>
</tr>
</tbody>
</table>
Environment Diagrams
Every expression is evaluated in the context of an environment
Environment Diagrams

- Every expression is evaluated in the context of an environment
- So far, the current environment is either:
Environment Diagrams

- Every expression is evaluated in the context of an environment
- So far, the current environment is either:
  - The global frame alone, or
Every expression is evaluated in the context of an environment

So far, the current environment is either:

- The global frame alone, or
- A local frame, followed by the global frame
Environment Diagrams

- Every expression is evaluated in the context of an environment
- So far, the current environment is either:
  - The global frame alone, or
  - A local frame, followed by the global frame

- **Important properties of environments:**
Every expression is evaluated in the context of an environment

So far, the current environment is either:

- The global frame alone, or
- A local frame, followed by the global frame

**Important properties of environments:**

- An environment is a sequence of frames
Environment Diagrams

- Every expression is evaluated in the context of an environment
- So far, the current environment is either:
  - The global frame alone, or
  - A local frame, followed by the global frame

**Important properties of environments:**
- An environment is a sequence of frames
- The earliest frame that contains a binding for a name determines the value that the name evaluates to
Every expression is evaluated in the context of an environment

So far, the current environment is either:
- The global frame alone, or
- A local frame, followed by the global frame

Important properties of environments:
- An environment is a sequence of frames
- The earliest frame that contains a binding for a name determines the value that the name evaluates to

The scope of a name is the region of code that has access to it
Environment of Function Application
Environment of Function Application

The environment in which a function is applied consists of:

it
Environment of Function Application

The environment in which a function is applied consists of:

- A new local frame \textit{each} time the function is \textit{applied}
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*
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

We refer to this as *lexical scoping*
Environment of Function Application

The environment in which a function is applied consists of:

it

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

- The environment in which the function was *defined*

- We refer to this as *lexical scoping*

- So far, this is 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*

- We refer to this as *lexical scoping*
- So far, this is just the global frame
- The *current* state of the environment is used, not the state when the function definition was executed
Formal Parameters

Example: [http://goo.gl/boCk0](http://goo.gl/boCk0)
def square(x):
    return mul(x, x)

Example: [link]
Formal Parameters

def square(x):
    return mul(x, x) VS

Example:  http://goo.gl/boCk0
Formal Parameters

```python
def square(x):
    return mul(x, x)

def square(y):
    return mul(y, y)
```

Example: [http://goo.gl/boCk0](http://goo.gl/boCk0)
**Formal Parameters**

```python
def square(x):
    return mul(x, x)

def square(y):
    return mul(y, y)
```

Example: [http://goo.gl/boCk0](http://goo.gl/boCk0)
Formal Parameters

```python
def square(x):
    return mul(x, x)

def square(y):
    return mul(y, y)
```

**Example:**

```
1 from operator import mul  
2 def square(x):  
3     return mul(x, x)  
4 square(-2)
```

**Global frame**

- `mul`
- `square`

**Variable frames**

- `x` in `square`
- `-2` in `square`
- `4` in `return`

Example: [http://goo.gl/boCk0](http://goo.gl/boCk0)
Multiple Environments in a Diagram

Example: [http://goo.gl/hrfnV](http://goo.gl/hrfnV)
Multiple Environments in a Diagram

What happens when to the local frame when a function returns?

Example: [http://goo.gl/hrfnV](http://goo.gl/hrfnV)
Multiple Environments in a Diagram

What happens when to the local frame when a function returns?

- It sticks around until Python realizes it is no longer needed

Example: [http://goo.gl/hrfnV](http://goo.gl/hrfnV)
Multiple Environments in a Diagram

What happens when to the local frame when a function returns?

- It sticks around until Python realizes it is no longer needed
- We will soon see cases where it is needed after the call

Example: [http://goo.gl/hrfnV](http://goo.gl/hrfnV)
Life Cycle of a User-Defined Function

Def statement:

```python
>>> def square(x):
    return mul(x, x)
```

What happens?

Call expression: `square(2+2)`

Calling/Applying: `square(x):`
Life Cycle of a User-Defined Function

Def statement:

- Name: square(x):
- Return expression: return mul(x, x)
- Body (return statement)

What happens?

Def statement:

Call expression: square(2+2)

Calling/Applying:

square(x):
Life Cycle of a User-Defined Function

Def statement:

Formal parameter

Name

square(x):

Return expression

Body (return statement)

Def statement: square(x):

return: mul(x, x)

What happens?

Function created

Call expression: square(2+2)

Calling/Applying:

square(x):
Life Cycle of a User-Defined Function

Def statement:

Formal parameter

Name

square(x):

Return expression

Body (return statement)

Def statement

What happens?

Function created

Name bound

Call expression:

square(2+2)

Calling/Applying:

square( x ):
Life Cycle of a User-Defined Function

Def statement: 
Formal parameter: square(x):
Return expression: return mul(x, x)
Body (return statement)

What happens?
Function created
Name bound

Def statement

Call expression: square(2+2)
operator: square
function: func square(x)
operand: 2+2
argument: 4

Calling/Applying:
square(x):
Life Cycle of a User-Defined Function

Def statement: square(x):

Formal parameter: x

Return expression: mul(x, x)

Body (return statement): return mul(x, x)

Call expression: square(2+2)

operator: square
function: func square(x)

operand: 2+2
argument: 4

Calling/Applying: square(x):

What happens?
Function created
Name bound
Op's evaluated
Life Cycle of a User-Defined Function

Def statement:

Formal parameter

Name

square(x):

Return expression

Body (return statement)

Def statement

Calling/Applying:

Call expression:

operator: square
function: func square(x)

square(2+2):

operand: 2+2
argument: 4

What happens?
Function created
Name bound
Op's evaluated
Function called with argument(s)
Life Cycle of a User-Defined Function

Def statement:
- Name: square(x)
- Body (return statement): return mul(x, x)

Call expression:
- Operator: square
- Function: func square(x)
- Argument: 4

What happens?
Function created
Name bound
Op's evaluated
Function called with argument(s)
Evaluates to return value below
Life Cycle of a User-Defined Function

Def statement:
- Name: square(x):
- Return expression: return mul(x, x)
- Body (return statement)

Call expression:
- Operator: square
- Function: func square(x)
- Operand: 2+2
- Argument: 4

Calling/Applying:
- Argument: 4
- Signature
- Return value: 16

What happens?
- Function created
- Name bound
- Op's evaluated
- Function called with argument(s)
- Evaluates to return value below
Life Cycle of a User-Defined Function

Def statement:
- Formal parameter: square(x):
  - Return expression: return mul(x, x)
  - Body (return statement)

Call expression:
- Operator: square
- Function: func square(x)
- Operand: 2+2
  - Argument: 4

What happens?
- Function created
- Name bound
- Op's evaluated
- Function called with argument(s)
- Evaluates to return value below
- New frame!
Life Cycle of a User-Defined Function

Def statement:
- Name: `square(x)`
- Return expression: `return: mul(x, x)`
- Body (return statement)

Call expression:
- Operator: `square` function: `func square(x)`
- Operand: `2+2`
- Argument: `4`

Calling/Applying:
- Argument: `4`
- Signature: `square(x)`
- Return value

What happens?
- Function created
- Name bound
- Op's evaluated
- Function called with argument(s)
- Evaluates to return value below
- New frame!
- Params bound
Life Cycle of a User-Defined Function

Def statement: Name: square(x):

Formal parameter: return: \text{mul}(x, x)

Body (return statement)

Call expression: square(2+2)

operator: square
function: func\ square(x)

operand: 2+2
argument: 4

Calling/Applying:

Argument: 4
Signature: square(x)

4 \rightarrow square(x):

16

Return value

What happens?

Function created
Name bound

Op's evaluated
Function called with argument(s)
Evaluates to return value below

New frame!
Params bound
Body executed
Python Feature Demonstration

- Operators
- Multiple Return Values
- Docstrings
- Doctests
- Default Arguments
- Statements
Statements
Statements

A *statement* is executed by the interpreter to perform an action.
A *statement* is executed by the interpreter to perform an action

Types of statements we have seen so far
A *statement* is executed by the interpreter to perform an action

Types of statements we have seen so far

- An assignment
A *statement* is executed by the interpreter to perform an action

Types of statements we have seen so far

- An assignment
  
  \[
  \text{radius} = 10
  \]
A *statement* is executed by the interpreter to perform an action

Types of statements we have seen so far

- An assignment
  - \texttt{radius = 10}

- A function definition
A *statement* is executed by the interpreter to perform an action

Types of statements we have seen so far

- An assignment
  
  \[\text{radius} = 10\]

- A function definition

  \[\text{def square}(x) :\]
A \textit{statement} is executed by the interpreter to perform an action.

Types of statements we have seen so far:

- An assignment
  \begin{verbatim}
  radius = 10
  \end{verbatim}

- A function definition
  \begin{verbatim}
  def square(x):
    return x * x
  \end{verbatim}
A statement is executed by the interpreter to perform an action

Types of statements we have seen so far

- An assignment
  
  \[
  \text{radius} = 10
  \]

- A function definition
  
  \[
  \text{def square}(x):
  \]
  
  \[
  \text{return } x \times x
  \]

- Returns, imports, assertions
Local Assignment

```python
1 def percent_difference(x, y):
2     difference = abs(x-y)
3     return 100 * difference / x
4     diff = percent_difference(40, 50)
```

Example: [http://goo.gl/1pyzL](http://goo.gl/1pyzL)
Local Assignment

Execution rule for assignment statements:

Example: [http://goo.gl/1pyzL](http://goo.gl/1pyzL)
Local Assignment

Execution rule for assignment statements:

1. Evaluate all expressions right of =, from left to right.
2. Bind the names on the left the resulting values in the first frame of the current environment.

Example: [http://goo.gl/1pyzL](http://goo.gl/1pyzL)
Compound Statements

A function definition is a *compound statement*
A function definition is a *compound statement*

Compound statements:

```
<header>:
  <statement>
  <statement>
  ...
<separating header>:
  <statement>
  <statement>
  ...
```

...
A function definition is a *compound statement*.

Compound statements:

```
<header>:
  <statement>
  <statement>
  ...
  <separating header>:
  <statement>
  <statement>
  ...
```

...
A function definition is a *compound statement*

Compound statements:

- **<header>:**
  - `<statement>`
  - `<statement>`
  - ...

- **<separating header>:**
  - `<statement>`
  - `<statement>`
  - ...

  ...
A function definition is a *compound statement*

**Compound statements:**

```plaintext
<header>:
  <statement>
  <statement>
  ...
</header>

<separating header>:
  <statement>
  <statement>
  ...
  ...
```
A function definition is a *compound statement*

**Compound statements:**
- `<header>`:
  - `<statement>`
  - `<statement>`
  - ...
- `<separating header>`:
  - `<statement>`
  - `<statement>`
  - ...
- `<statement>`
  - `<statement>`
  - ...

The first header determines a statement’s type
A function definition is a *compound statement*.

Compound statements:

```
<header>:
  <statement>
  <statement>
  ...
<separating header>:
  <statement>
  <statement>
  ...
```

The first header determines a statement’s type.

The header of a clause “controls” the suite that follows.
Compound Statements

Compound statements:

<header>:</header>:
<statement> <statement> <statement>
Suite
...

<separating header>:</separating header>:
<statement> <statement> <statement> <statement> <statement>
...
...

Compound statements:

<header>:
  <statement>
  <statement>
  ...
</header>

A suite is a sequence of statements

<separating header>:
  <statement>
  <statement>
  ...
  ...

Suite
Compound Statements

Compound statements:

<header>:
    <statement>
    <statement>
    ...
    Suite
 <separating header>:
    <statement>
    <statement>
    ...
    ...

A suite is a sequence of statements

To “execute” a suite means to execute its sequence of statements, in order
Compound Statements

Compound statements:

<header>:
  <statement>
  Suite
  <statement>
...<statement>

<separating header>:
  <statement>
  <statement>
...<statement>

...<statement>

A suite is a sequence of statements

To “execute” a suite means to execute its sequence of statements, in order

Execution rule for a sequence of statements:

1. Execute the first
2. Unless directed otherwise, execute the rest
def absolute_value(x):
    
    """Return the absolute value of x."""
    if x > 0:
        return x
    elif x == 0:
        return 0
    else:
        return -x
def absolute_value(x):
    """Return the absolute value of x."""
    if x > 0:
        return x
    elif x == 0:
        return 0
    else:
        return -x
Conditional Statements

```
def absolute_value(x):
    """Return the absolute value of x."""
    if x > 0:
        return x
    elif x == 0:
        return 0
    else:
        return -x
```

Execution rule for conditional statements:
Conditional Statements

```python
def absolute_value(x):
    """Return the absolute value of x."""
    if x > 0:
        return x
    elif x == 0:
        return 0
    else:
        return -x
```

1 statement,
3 clauses,
3 headers,
3 suites

Execution rule for conditional statements:

Each clause is considered in order.
1. Evaluate the header's expression.
2. If it is a true value, execute the suite & skip the remaining clauses.
def absolute_value(x):
    """Return the absolute value of x.""
    if x > 0:
        return x
    elif x == 0:
        return 0
    else:
        return -x
def absolute_value(x):
    """Return the absolute value of x."""
    if x > 0:
        return x
    elif x == 0:
        return 0
    else:
        return -x
def absolute_value(x):
    """Return the absolute value of x."""
    if x > 0:
        return x
    elif x == 0:
        return 0
    else:
        return -x

George Boole
def absolute_value(x):
    """Return the absolute value of x."""
    if x > 0:
        return x
    elif x == 0:
        return 0
    else:
        return -x

Boolean Contexts

George Boole

Two boolean contexts
def absolute_value(x):
    """Return the absolute value of x."""
    if x > 0:
        return x
    elif x == 0:
        return 0
    else:
        return -x
def absolute_value(x):
  """Return the absolute value of x."""
  if x > 0:
    return x
  elif x == 0:
    return 0
  else:
    return -x

Two boolean contexts

False values in Python: False, 0, "", None
Boolean Contexts

```python
def absolute_value(x):
    """Return the absolute value of x.""
    if x > 0:
        return x
    elif x == 0:
        return 0
    else:
        return -x
```

Two boolean contexts

George Boole

False values in Python: False, 0, "", None (more to come)
def absolute_value(x):
    """Return the absolute value of x."""
    if x > 0:
        return x
    elif x == 0:
        return 0
    else:
        return -x

Boolean Contexts

Two boolean contexts

George Boole

False values in Python: False, 0, "", None (more to come)

True values in Python: Anything else (True)
def absolute_value(x):
    """Return the absolute value of x."""
    if x > 0:
        return x
    elif x == 0:
        return 0
    else:
        return -x

Boolean Contexts

George Boole

False values in Python: False, 0, "", None (more to come)

True values in Python: Anything else (True)

Read Section 1.5.4!
Iteration

Execution rule for while statements:

1. Evaluate the header’s expression.
2. If it is a true value, execute the (whole) suite, then return to step 1.

Example: [http://goo.gl/mk7Sc](http://goo.gl/mk7Sc)
Iteration

Execution rule for while statements:

1. Evaluate the header’s expression.
2. If it is a true value, execute the (whole) suite, then return to step 1.

Example: [http://goo.gl/mk7Sc](http://goo.gl/mk7Sc)
Iteration

Execution rule for while statements:

1. Evaluate the header’s expression.
2. If it is a true value, execute the (whole) suite, then return to step 1.

Example: [http://goo.gl/mk7Sc](http://goo.gl/mk7Sc)
Iteration

Example:

```
i, total = 0, 0
while i < 3:
    i = i + 1
    total = total + i
```

Execution rule for while statements:

1. Evaluate the header’s expression.
2. If it is a true value, execute the (whole) suite, then return to step 1.

Example: [http://goo.gl/mk7Sc](http://goo.gl/mk7Sc)
Execution rule for while statements:

1. Evaluate the header’s expression.
2. If it is a true value, execute the (whole) suite, then return to step 1.

Example: [http://goo.gl/mk7Sc](http://goo.gl/mk7Sc)
Iteration

Execution rule for while statements:

1. Evaluate the header’s expression.
2. If it is a true value, execute the (whole) suite, then return to step 1.

Example: [http://goo.gl/mk7Sc](http://goo.gl/mk7Sc)
Iteration

Execution rule for while statements:

1. Evaluate the header’s expression.
2. If it is a true value, execute the (whole) suite, then return to step 1.

Example: [http://goo.gl/mk7Sc](http://goo.gl/mk7Sc)
Iteration

**Execution rule for while statements:**

1. Evaluate the header’s expression.
2. If it is a true value,
   execute the (whole) suite,
   then return to step 1.

Example: [http://goo.gl/mk7Sc](http://goo.gl/mk7Sc)
Execution rule for while statements:

1. Evaluate the header’s expression.
2. If it is a true value, execute the (whole) suite, then return to step 1.

Example: [http://goo.gl/mk7Sc](http://goo.gl/mk7Sc)
Iteration

```
>>> i, total = 0, 0
>>> while i < 3:
    >>> i = i + 1
    >>> total = total + i
```

Global frame

- i: 2
- total: 1

Execution rule for while statements:

1. Evaluate the header’s expression.
2. If it is a true value, execute the (whole) suite, then return to step 1.

Example: [http://goo.gl/mk7Sc](http://goo.gl/mk7Sc)
Execution rule for while statements:

1. Evaluate the header’s expression.
2. If it is a true value, execute the (whole) suite, then return to step 1.

Example: [http://goo.gl/mk7Sc](http://goo.gl/mk7Sc)
Iteration

Execution rule for while statements:

1. Evaluate the header’s expression.
2. If it is a true value, execute the (whole) suite, then return to step 1.

Example: [http://goo.gl/mk7Sc](http://goo.gl/mk7Sc)
Execution rule for while statements:

1. Evaluate the header’s expression.
2. If it is a true value, execute the (whole) suite, then return to step 1.

Example: [http://goo.gl/mk7Sc](http://goo.gl/mk7Sc)
Execution rule for while statements:

1. Evaluate the header’s expression.
2. If it is a true value, execute the (whole) suite, then return to step 1.

Example: [http://goo.gl/mk7Sc](http://goo.gl/mk7Sc)
Iteration

Execution rule for while statements:

1. Evaluate the header’s expression.
2. If it is a true value, execute the (whole) suite, then return to step 1.

Example: http://goo.gl/mk7Sc
Iteration

Execution rule for while statements:

1. Evaluate the header’s expression.
2. If it is a true value, execute the (whole) suite, then return to step 1.

Example: [http://goo.gl/mk7Sc](http://goo.gl/mk7Sc)
Iteration

Execution rule for while statements:

1. Evaluate the header’s expression.
2. If it is a true value, execute the (whole) suite, then return to step 1.

Example: [http://goo.gl/mk7Sc](http://goo.gl/mk7Sc)
Iteration

Execution rule for while statements:

1. Evaluate the header’s expression.
2. If it is a true value, execute the (whole) suite, then return to step 1.

Example: [http://goo.gl/mk7Sc](http://goo.gl/mk7Sc)
Locally Defined Functions
Functions can be defined inside other functions
def sum_of_squares(n):
    """Sum of the squares of the integers 1 to n""
    def square(x):
        return mul(x, x)
    total, k = 0, 1
    while k <= n:
        total, k = total + square(k), k + 1
    return total
Functions can be defined inside other functions

What happens when a def is executed?

def sum_of_squares(n):
    """Sum of the squares of the integers 1 to n""
    def square(x):
        return mul(x, x)
    total, k = 0, 1
    while k <= n:
        total, k = total + square(k), k + 1
    return total
Locally Defined Functions

Functions can be defined inside other functions

What happens when a def is executed?
1. Create a function value with the given signature and body

```python
def sum_of_squares(n):
    """Sum of the squares of the integers 1 to n""
    def square(x):
        return mul(x, x)
    total, k = 0, 1
    while k <= n:
        total, k = total + square(k), k + 1
    return total
```
Functions can be defined inside other functions

What happens when a def is executed?
1. Create a function value with the given signature and body
2. Bind the given name to that value in the current frame

```python
def sum_of_squares(n):
    """Sum of the squares of the integers 1 to n""
    def square(x):
        return mul(x, x)
    total, k = 0, 1
    while k <= n:
        total, k = total + square(k), k + 1
    return total
```
Functions can be defined inside other functions

What happens when a def is executed?
1. Create a function value with the given signature and body
2. Bind the given name to that value in the current frame

The name can then be used to call the function.

```python
def sum_of_squares(n):
    """Sum of the squares of the integers 1 to n""
    def square(x):
        return mul(x, x)
    total, k = 0, 1
    while k <= n:
        total, k = total + square(k), k + 1
    return total
```
Locally Defined Functions

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

Example: [http://goo.gl/pnU8f](http://goo.gl/pnU8f)
```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

```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
```
Functions as Return Values

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_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

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

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

A function that returns a function

A local def statement
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."
    def adder(k):
        return add(n, k)
    return adder

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

A function that returns a function

The name `add_three` is bound to a function

A local `def` statement
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.""
    def adder(k):
        return add(n, k)
    return adder
```

A function that returns a function

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

The name add_three is bound to a function

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

A local def statement

Can refer to names in the enclosing function
def make_adder(n):
    def adder(k):
        return add(n, k)
    return adder

make_adder(1)(2)
Call Expressions as Operators

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

make_adder(1)(2)
make_adder(1)(2)
```

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

```python
def make_adder(n):
    def adder(k):
        return add(n, k)
    return adder
```
Call Expressions as Operators

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

make_adder(1)(2)
```

Operator

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

```python
make_adder(1)(2)
```
Call Expressions as Operators

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

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

**Operator**

**Operand 0**

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

```python
def make_adder(n):
def adder(k):
    return add(n, k)
return adder
```
Call Expressions as Operators

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

make_adder(1)(2)
```

An expression that evaluates to a function value

```
def make_adder(n):
    def adder(k):
        return add(n, k)
    return adder
```
Call Expressions as Operators

```
def make_adder(n):
    def adder(k):
        return add(n, k)
    return adder
make_adder(1)(2)
```

An expression that evaluates to a function value

An expression that evaluates to any value