61A Lecture 3

Wednesday, August 31
Lightning Review: Expressions

Primitive expressions:

Call expressions:
Lightning Review: Expressions

Primitive expressions: 2

Call expressions:
Lightning Review: Expressions

Primitive expressions:  2  add

Call expressions:
Lightning Review: Expressions

Primitive expressions: 2

Call expressions:
Lightning Review: Expressions

Primitive expressions:  
- 2
- add
- 'hello'

Number | Name | String

Call expressions:

```
add ( 2 , 3 )
```
**Lightning Review: Expressions**

**Primitive expressions:**
- Number: 2
- Name: add
- String: 'hello'

**Call expressions:**
```
add ( 2 , 3 )
```

*Operator*
Lightning Review: Expressions

Primitive expressions:

- 2
- add
- 'hello'

Call expressions:

- add
- ( 2 , 3 )
Lightning Review: Expressions

**Primitive expressions:**
- Number: 2
- Name: add
- String: 'hello'

**Call expressions:**
- `add(2, add(3, 5))`

*One big nested call expression*

```
mul(add(2, mul(4, 6)), add(3, 5))
```
Life Cycle of a User-Defined Function

Defining:

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

Call expression: `square(2+2)`

Calling/Applying:

```
square(x):
    return mul(x, x)
```
Life Cycle of a User-Defined Function

Defining:

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

Def statement

Call expression:  square(2+2)

 Calling/Applying:

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

Defining:

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

Call expression: `square(2+2)`

Calling/Applying:

```python
square(x):
    return mul(x, x)
```
Life Cycle of a User-Defined Function

Defining:

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

Call expression: `square(2+2)`

Calling/Applying:

```python
square(x):
    return mul(x, x)
```
Life Cycle of a User-Defined Function

Defining:

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

Call expression: `square(2+2)`

Calling/Applying:

```python
square( x ):
    return mul(x, x)
```
Life Cycle of a User-Defined Function

Defining:

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

Call expression: `square(2+2)`

Calling/Applying:

```python
square(x):
    return mul(x, x)
```
Life Cycle of a User-Defined Function

Defining:

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

Call expression:

`square(2+2)`

Calling/Applying:

```python
square(x):
    return mul(x, x)
```
Life Cycle of a User-Defined Function

Defining:

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

Def statement

Formal parameter

Return expression

Body (return statement)

What happens?
Function created
Body stored

Call expression: square(2+2)

Calling/Applying:

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

Defining:

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

What happens?

Function created
Body stored
Name bound

Call expression: square(2+2)

Calling/Applying:

    def square(x):
    return mul(x, x)
Life Cycle of a User-Defined Function

Defining:

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

- Def statement
- Formal parameter: x
- Return expression: return mul(x, x)
- Body (return statement)
- Name bound
- Function created
- Body stored
- What happens?

Call expression:

```
square(2+2)
```

- Operand: 2+2
- Argument: 4

Calling/Applying:

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

Wednesday, August 31, 2011
Life Cycle of a User-Defined Function

Defining:

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

Call expression:

operator: square
function: square

Calling/Applying:

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

What happens?
Function created
Body stored
Name bound
Life Cycle of a User-Defined Function

Defining:

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

- **Def statement**
- **Formal parameter** `x`
- **Return expression** `mul(x, x)`
- **Body** `(return statement)`
- **Function created**
- **Body stored**
- **Name bound**

Call expression:

- **Operator**: `square`
- **Function**: `square`
- **Operand**: `2+2`
- **Argument**: `4`

What happens?

- **Op's evaluated**
- **Function created**
- **Body stored**
- **Name bound**

Calling/Applying:

```python
square(x):
    return mul(x, x)
```
Life Cycle of a User-Defined Function

Defining:
>>> def square(x):
    return mul(x, x)

What happens?
Function created
Body stored
Name bound

Call expression:
operator: square
function: square

Operand: 2 + 2
Argument: 4

Calling/Applying:

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

**Defining:**

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

- **Formal parameter:** `x`
- **Return expression:** `return mul(x, x)`
- **Body:** `return mul(x, x)`
- **Def statement**
- **Function created**
- **Body stored**
- **Name bound**

**Call expression:**

- **Operator:** `square`
- **Function:** `square`
- **Operand:** `2+2`
- **Argument:** `4`

**Calling/Applying:**

```python
square(2+2)
```

- **Intrinsic name**
- **Op's evaluated**
- **Function called**
Life Cycle of a User-Defined Function

Defining:

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

Def statement

Formal parameter

Return expression

Body (return statement)

What happens?

Function created
Body stored
Name bound

Call expression:

operator: square
function: square

operand: 2+2
argument: 4

Op's evaluated
Function called

Calling/Applying:

Signature

Intrinsic name

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

Defining:

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

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

Call expression:

- **operator**: `square`
- **function**: `square`
- **operand**: `2+2`
- **argument**: `4`

Calling/Applying:

```python
4 ➔ square(x):
```

- **Signature**:
- **Intrinsic name**: `square`
- **Return**
- **Expression**

What happens?

- Function created
- Body stored
- Name bound

Op's evaluated

Function called
Life Cycle of a User-Defined Function

Defining:

Def statement

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

Def expression

Body (return statement)

Calling/Applying:

Call expression:

operator: square
function: square

operand: 2+2
argument: 4

What happens?

Function created
Body stored
Name bound

Op's evaluated
Function called

Signature

Intrinsic name

4 \texttt{square}(x):

\texttt{return mul}(x, x) \rightarrow 16
Life Cycle of a User-Defined Function

Defining:

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

Calling/Applying:

operator: square
function: square
operand: 2+2
argument: 4

What happens?

Function created
Body stored
Name bound

Op's evaluated
Function called
Life Cycle of a User-Defined Function

Defining:

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

Def statement
Formal parameter
Return expression
Body (return statement)

Call expression:

Operator: square
Function: square
Operand: 2+2
Argument: 4

What happens?
Function created
Body stored
Name bound

Op's evaluated
Function called

Calling/Applying:

Argument
Intrinsic name

Signature
Return value
Life Cycle of a User-Defined Function

Defining:

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

What happens?
Function created
Body stored
Name bound

Call expression:
operator: square
function: square

Op's evaluated
Function called

Calling/Applying:
Argument
Intrinsic name

New frame!
Life Cycle of a User-Defined Function

Defining:

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

What happens?
Function created
Body stored
Name bound

Calling/Applying:

```
square(2+2)
```

Op's evaluated
Function called

New frame!
Params bound

---

Wednesday, August 31, 2011
Life Cycle of a User-Defined Function

Defining:

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

What happens?
- Function created
- Body stored
- Name bound

Call expression:

- Operator: `square`
- Function: `square`
- Operand: 2+2
- Argument: 4

Calling/Applying:

- Argument: 4
- Intrinsic name
- Signature
- Body evaluated
- Return value
- New frame!
- Params bound
Cast of Characters: Environment Diagrams

Frames:

a: \rightarrow 2
b: \rightarrow 5

Environments:
Cast of Characters: Environment Diagrams

Frames:

Environments:
Cast of Characters: Environment Diagrams

Frames:

```
Name

<table>
<thead>
<tr>
<th>a:</th>
<th>→ 2: Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>b:</td>
<td>→ 5</td>
</tr>
</tbody>
</table>
```

Environments:
Cast of Characters: Environment Diagrams

Frames:

Environments:
Frames:
A name is bound to a value

Environments:
Cast of Characters: Environment Diagrams

Frames:
A name is bound to a value

Environments:
Frames:
A name is bound to a value
A frame is a rectangle that contains bindings

Environments:
Frames:
A name is bound to a value

A frame is a rectangle that contains bindings

In a frame, there is at most one binding per name

Environments:
Frames:

A name is bound to a value

A frame is a rectangle that contains bindings

In a frame, there is at most one binding per name

Environments:

---

Wednesday, August 31, 2011
Frames:
A name is bound to a value
A frame is a rectangle that contains bindings
In a frame, there is at most one binding per name

Environments:
Frames:

A name is bound to a value.

A frame is a rectangle that contains bindings.

In a frame, there is at most one binding per name.

Environments:

An environment is a sequence of frames.
Frames:
A name is bound to a value
A frame is a rectangle that contains bindings
In a frame, there is at most one binding per name

Environments:
An environment is a sequence of frames
So far, environments only have at most two frames
Frames:
A name is bound to a value
A frame is a rectangle that contains bindings
In a frame, there is at most one binding per name

Environments:
An environment is a sequence of frames
So far, environments only have at most two frames
Frames:

A name is bound to a value

A frame is a rectangle that contains bindings

In a frame, there is at most one binding per name

Environments:

An environment is a sequence of frames

So far, environments only have at most two frames
Frames:
A name is bound to a value

A frame is a rectangle that contains bindings

In a frame, there is at most one binding per name

Environments:
An environment is a sequence of frames

So far, environments only have at most two frames
(Friday: longer sequences)
An Environment is a Sequence of Frames

Environments (Memory):

The global frame

Local frame
An Environment is a Sequence of Frames

Environments (Memory):
Frames link to each other
An Environment is a Sequence of Frames

Environments (Memory):
- Frames link to each other
- An environment is a sequence of frames
An Environment is a Sequence of Frames

Environments (Memory):

Frames link to each other

An environment is a sequence of frames

An environment is a first frame, plus the frames that follow
An Environment is a Sequence of Frames

Environments (Memory):

Frames link to each other

An environment is a sequence of frames

An environment is a first frame, plus the frames that follow
An Environment is a Sequence of Frames

Environments (Memory):
Frames link to each other

An environment is a sequence of frames

An environment is a first frame, plus the frames that follow

An environment is a first frame, plus the sequence of frames that follow
An Environment is a Sequence of Frames

Environments (Memory):

Frames link to each other

An environment is a sequence of frames

An environment is a first frame, plus the frames that follow

An environment is a first frame, plus the sequence of frames that follow
An Environment is a Sequence of Frames

Environments (Memory):

Frames link to each other

An environment is a sequence of frames

An environment is a first frame, plus the frames that follow

An environment is a first frame, plus the sequence of frames that follow
An Environment is a Sequence of Frames

Environments (Memory):

Frames link to each other

An environment is a sequence of frames

An environment is a first frame, plus the frames that follow

An environment is a first frame, plus the sequence of frames that follow

An environment is a first frame, plus the environment that follows
An Environment is a Sequence of Frames

Environments (Memory):

- Frames link to each other
- An environment is a sequence of frames
- An environment is a first frame, plus the frames that follow
- An environment is a first frame, plus the sequence of frames that follow
- An environment is a first frame, plus the environment that follows
**Environments (Memory):**

Frames link to each other

An environment is a sequence of frames

An environment is a first frame, plus the frames that follow

An environment is a first frame, plus the sequence of frames that follow

An environment is a first frame, plus the environment that follows
An Environment is a Sequence of Frames

Environments (Memory):

Frames link to each other
An environment is a sequence of frames
An environment is a first frame, plus the frames that follow
An environment is a first frame, plus the sequence of frames that follow
An environment is a first frame, plus the environment that follows

Existing environment
Extends
First frame
New environment

x: ...
Local frame

mul: ...
The global frame
An Expression is Evaluated in an Environment

Environments (Memory):
Frames link to each other
An environment is a sequence of frames
An environment is a first frame, plus the environment that follows

Expressions (Program):
An Expression is Evaluated in an Environment

Environments (Memory):
Frames link to each other
An environment is a sequence of frames
An environment is a first frame, plus the environment that follows

Expressions (Program):
Expressions are Python code
An Expression is Evaluated in an Environment

Environments (Memory):
- Frames link to each other

An environment is a *sequence* of frames

Expressions (Program):
- Expressions are Python code

```python
square(2)
return mul(x, x)
```
An Expression is Evaluated in an Environment

Environments (Memory):
Frames link to each other
An environment is a sequence of frames
An environment is a first frame, plus the environment that follows

Expressions (Program):
Expressions are Python code

Example:
```
square(2)
```
```
return mul(x, x)
```

The global frame
Local frame
x:
mul:
An Expression is Evaluated in an Environment

Environments (Memory):
- Frames link to each other
- An environment is a sequence of frames
- An environment is a first frame, plus the environment that follows

Expressions (Program):
- Expressions are Python code

---

Wednesday, August 31, 2011
An Expression is Evaluated in an Environment

Environments (Memory):
Frames link to each other

An environment is a sequence of frames

An environment is a first frame, plus the environment that follows

Expressions (Program):
Expressions are Python code

---

Wednesday, August 31, 2011
An Expression is Evaluated in an Environment

Environments (Memory):
- Frames link to each other
- An environment is a sequence of frames
- An environment is a first frame, plus the environment that follows

Expressions (Program):
- Expressions are Python code
- *Not* part of an environment
An Expression is Evaluated in an Environment

Environments (Memory):
Frames link to each other
An environment is a sequence of frames
An environment is a first frame, plus the environment that follows

Expressions (Program):
Expressions are Python code
Not part of an environment
They are evaluated in an environment to yield a value
Multiple Environments in One Diagram!

```
from operator import mul

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

square(square(square(3)))
```
Every call to a user-defined function creates a new local frame.

```
from operator import mul
def square(x):
    return mul(x, x)
```

```
square(square(square(3)))
```
Multiple Environments in One Diagram!

Every call to a user-defined function creates a new local frame.

```
from operator import mul
def square(x):
    return mul(x, x)
square(square(3))
```
Multiple Environments in One Diagram!

Every call to a user-defined function creates a new local frame.

```
from operator import mul

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

def mul(a, b):
    return mul(a, b)

mul:
    ...

square:

square(square(3))
```

```
from operator import mul
def square(x):
    return mul(x, x)
square(square(square(3)))
```
Multiple Environments in One Diagram!

```
from operator import mul

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

square(square(3))
```

Every call to a user-defined function creates a new local frame.
Multiple Environments in One Diagram!

```
from operator import mul

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

square(square(3))
```

Every call to a user-defined function creates a new local frame.
Multiple Environments in One Diagram!

```
from operator import mul

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

square(square(3))
```

Every call to a user-defined function creates a new local frame.
Multiple Environments in One Diagram!

Every call to a user-defined function creates a new local frame

```
from operator import mul

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

square(square(3))
```

```
9
```

```
square(square(3))
```
Multiple Environments in One Diagram!

Every call to a user-defined function creates a new local frame

\[
\text{square(3)} 
\rightarrow \text{square} 
\rightarrow \text{mul(3, 3)} 
\rightarrow \text{mul(a, b)}: 
\rightarrow \text{square(x)}: 
\rightarrow \text{return mul(x, x)} 
\]

from operator import mul
def square(x):
    return mul(x, x)

square(square(square(3)))
Multiple Environments in One Diagram!

Every call to a user-defined function creates a new local frame.

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

```python
from operator import mul
def square(x):
    return mul(x, x)
square(square(3))
```

Wednesday, August 31, 2011
Multiple Environments in One Diagram!

Every call to a user-defined function creates a new local frame

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

from operator import mul

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

square(square(3))
```

Wednesday, August 31, 2011
Names Have No Meaning Without Environments

from operator import mul
def square(x):
    return mul(x, x)
square(-2)
Names Have No Meaning Without Environments

```python
from operator import mul

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

square(-2)
```
Names Have No Meaning Without Environments

A name evaluates to the value bound to that name

```python
from operator import mul
def square(x):
    return mul(x, x)
square(-2)
```

Wednesday, August 31, 2011
Names Have No Meaning Without Environments

A name evaluates to the value bound to that name

...in the earliest frame of the current environment

from operator import mul
def square(x):
    return mul(x, x)
square(-2)
A name evaluates to the value bound to that name

...in the earliest frame of the current environment

...in which that name is found

from operator import mul
def square(x):
    return mul(x, x)
square(-2)
Names Have No Meaning Without Environments

A name evaluates to the **value bound** to that name

...in the **earliest** frame of the **current environment**

...in which that name is found

```python
from operator import mul
def square(x):
    return mul(x, x)
square(-2)
```

Wednesday, August 31, 2011
A name evaluates to the value bound to that name

...in the earliest frame of the current environment

...in which that name is found

from operator import mul
def square(x):
    return mul(x, x)
square(-2)
A name evaluates to the value bound to that name...
in the earliest frame of the current environment...
in which that name is found

from operator import mul
def square(x):
    return mul(x, x)
square(-2)
A name evaluates to the value bound to that name

...in the earliest frame of the current environment

...in which that name is found

from operator import mul
def square(x):
    return mul(x, x)
square(-2)
Formal Parameters
**Formal Parameters**

def square(x):
    return mul(x, x)
Formal Parameters

def square(x):
    return mul(x, x) vs
Formal Parameters

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

vs

def square(y):
    return mul(y, y)
Formal Parameters

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

vs

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

Formal parameter names stay local to their frame
def square(x):
    return mul(x, x)

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

Formal Parameters

mul:
...
square:

mul(a,b):

square(__):

return mul(__, __)

Formal parameter names stay local to their frame
Formal Parameters

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

vs

```
from operator import mul
def square(__):
    return mul(__, __)
square(-2)
```

Formal parameter names stay local to their frame
Formal Parameters

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

vs

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

from operator import mul

def square(__):
    return mul(__, __)

square(-2)

Local frame

mul:
...

square:

__: -2

mul(a, b):

square(__):
    return mul(__, __)

from operator import mul

def square(__):
    return mul(__, __)

square(-2)
Shadowing Names

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

square(-2)
def square(mul):
    return mul(mul, mul)

square(-2)

If we use this formal parameter
def square(mul):
    return mul(mul, mul)

square(-2)
def square(mul):
    return mul(mul, mul)

square(-2)

If we use this formal parameter

mul:
...
square:

mul: -2

square:

mul(a,b):

square(mul):
    return mul(mul, mul)

Evaluating this call expression will fail

Careful!
def square(mul):
    return mul(mul, mul)

square(-2)

mul:
...
square:

mul: -2

square:

mul(a,b):

square(mul):
    return mul(mul, mul)

square(-2)

return mul(mul, mul)

If we use this formal parameter

Evaluating this call expression will fail

Careful!
Python Feature Demonstration

<Demo>

Operators

Multiple Return Values

Docstrings

Doctests

Default Arguments

Statements

</Demo>
A statement is executed by the interpreter to perform an action.
A statement is executed by the interpreter to perform an action

Compound statements:

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

Wednesday, August 31, 2011
A statement is executed by the interpret to perform an action

Compound statements:

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

Wednesday, August 31, 2011
A statement is executed by the interpret to perform an action

Compound statements:

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

Wednesday, August 31, 2011
A statement is executed by the interpret to perform an action

**Compound statements:**

- **Statement**
  - **Clause**
  - **Suite**

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

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

  Wednesday, August 31, 2011
A statement
is executed by the interpret
to perform an action

**Compound statements:**

The first header
determines a statement’s type
A statement
is executed by the interpret
to perform an action

Compound statements:

The first header
determines a
statement’s type

The header of a clause
“controls” the suite
that follows
A statement is executed by the interpreter to perform an action.

**Compound statements:**

- The first header determines a statement's type.
- The header of a clause "controls" the suite that follows.
- def statements are compound statements.
Compound Statements

Compound statements:

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

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

Suite
Compound statements:

A suite is a sequence of statements

Suite
Compound Statements

Compound statements:

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>`
  - `<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

**Execution Rule for a sequence of statements:**

- Execute the first
- Unless directed otherwise, execute the rest
def percent_difference(x, y):
    difference = abs(x-y)
    return 100 * difference / x
percent_difference(40, 50)
def percent_difference(x, y):
    difference = abs(x - y)
    return 100 * difference / x

percent_difference(40, 50)
Assignment binds names in the *first frame* of the current environment.

```
def percent_difference(x, y):
    difference = abs(x-y)
    return 100 * difference / x

percent_difference(40, 50)
```
Local Assignment

Assignment binds names in the **first frame** of the current environment

```
def percent_difference(x, y):
    difference = abs(x-y)
    return 100 * difference / x

percent_difference(40, 50)
```

Assignment binds names in the **first frame** of the current environment
def percent_difference(x, y):
    difference = abs(x - y)
    return 100 * difference / x

percent_difference(40, 50)

Assignment binds names in the *first frame* of the current environment.
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

Execution rule for conditional statements:
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:

Each clause is considered in order.

1. Evaluate the header's expression.

2. If it is a true value, execute the suite & skip 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
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
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

False values in Python: False, 0, ‘’, None
```
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 (more to come)
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

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

George Boole

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

True values in Python: Anything else (True)

Read Section 1.5.4!
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.
Iteration

```
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.
Iteration

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.
Iteration

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

<table>
<thead>
<tr>
<th>i</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

**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.
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.
Iteration

```python
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.
Evaluation 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.
Iteration

```python
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.
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.
### 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.

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

```python
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.
Iteration

\[ i, \text{total} = 0, 0 \]

\[ \text{while } i < 3: \]
\[ i = i + 1 \]
\[ \text{total} = \text{total} + i \]

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.
Iteration

```
\[
\begin{align*}
\text{i, total} &= 0, 0 \\
\text{while } i < 3: \\
\quad \text{i} &= \text{i} + 1 \\
\quad \text{total} &= \text{total} + \text{i}
\end{align*}
\]
```

**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.
**Iteration**

```python
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.
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.
Iteration

```python
>>> 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.
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.
Iteration

```python
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.
**Iteration**

```python
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.
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.
Iteration

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

0, 1, 1, 2, 3, 5, 8, 13, ...
The Fibonacci Sequence

0, 1, 1, 2, 3, 5, 8, 13, ...

def fib(n):
    """Compute the nth Fibonacci number, for n >= 2."""
    pred, curr = 0, 1  # First two Fibonacci numbers
    k = 2  # Tracks which Fib number is curr
    while k < n:
        pred, curr = curr, pred + curr
        k = k + 1
    return curr
The Fibonacci Sequence

```
def fib(n):
    """Compute the nth Fibonacci number, for n >= 2."""
    pred, curr = 0, 1  # First two Fibonacci numbers
    k = 2  # Tracks which Fib number is curr
    while k < n:
        pred, curr = curr, pred + curr
        k = k + 1
    return curr
```
The Fibonacci Sequence

```python
def fib(n):
    """Compute the nth Fibonacci number, for n >= 2."""
    pred, curr = 0, 1  # First two Fibonacci numbers
    k = 2  # Tracks which Fib number is curr
    while k < n:
        pred, curr = curr, pred + curr
        k = k + 1
    return curr
```

... 0, 1, 1, 2, 3, 5, 8, 13, ...

... pred: curr:
The Fibonacci Sequence

def fib(n):
    """Compute the nth Fibonacci number, for n >= 2."""
    pred, curr = 0, 1  # First two Fibonacci numbers
    k = 2  # Tracks which Fib number is curr
    while k < n:
        pred, curr = curr, pred + curr
        k = k + 1
    return curr
The Fibonacci Sequence

def fib(n):
    """Compute the nth Fibonacci number, for n >= 2."""
    pred, curr = 0, 1  # First two Fibonacci numbers
    k = 2  # Tracks which Fib number is curr
    while k < n:
        pred, curr = curr, pred + curr
        k = k + 1
    return curr
The Fibonacci Sequence

```python
def fib(n):
    """Compute the nth Fibonacci number, for n >= 2."""
    pred, curr = 0, 1  # First two Fibonacci numbers
    k = 2  # Tracks which Fib number is curr
    while k < n:
        pred, curr = curr, pred + curr
        k = k + 1
    return curr
```

0, 1, 1, 2, 3, 5, 8, 13, ...

...
The Fibonacci Sequence

def fib(n):
    """Compute the nth Fibonacci number, for n >= 2."""
    pred, curr = 0, 1  # First two Fibonacci numbers
    k = 2  # Tracks which Fib number is curr
    while k < n:
        pred, curr = curr, pred + curr
        k = k + 1
    return curr
The Fibonacci Sequence

```
def fib(n):
    """Compute the nth Fibonacci number, for n >= 2."""
    pred, curr = 0, 1  # First two Fibonacci numbers
    k = 2              # Tracks which Fib number is curr
    while k < n:
        pred, curr = curr, pred + curr
        k = k + 1
    return curr
```

0, 1, 1, 2, 3, 5, 8, 13, ...

...
Project 1: Pig

(Demo)