Announcements
Print and None

(Demo)
None Indicates that Nothing is Returned
None Indicates that Nothing is Returned

The special value \texttt{None} represents nothing in Python.
None Indicates that Nothing is Returned

The special value `None` represents nothing in Python.

A function that does not explicitly return a value will return `None`.
None Indicates that Nothing is Returned

The special value `None` represents nothing in Python.

A function that does not explicitly return a value will return `None`.

*Careful:* `None` is *not displayed* by the interpreter as the value of an expression.
None Indicates that Nothing is Returned

The special value `None` represents nothing in Python.

A function that does not explicitly return a value will return `None`.

*Careful:* `None` is *not displayed* by the interpreter as the value of an expression.

```python
>>> def does_not_return_square(x):
...     ...     x * x
...     ...
...     ...
```
None Indicates that Nothing is Returned

The special value `None` represents nothing in Python.

A function that does not explicitly return a value will return `None`.

_Careful:_ `None` is _not displayed_ by the interpreter as the value of an expression.

```python
>>> def does_not_return_square(x):
...     x * x
...     No return
```
None Indicates that Nothing is Returned

The special value `None` represents nothing in Python.

A function that does not explicitly return a value will return `None`.

*Careful*: `None` is *not displayed* by the interpreter as the value of an expression.

```python
>>> def does_not_return_square(x):
...     x * x
...     # No return

>>> does_not_return_square(4)
```
None Indicates that Nothing is Returned

The special value `None` represents nothing in Python.

A function that does not explicitly return a value will return `None`.

*Careful:* `None` is *not displayed* by the interpreter as the value of an expression.

```python
>>> def does_not_return_square(x):
...     x * x
... >>> does_not_return_square(4)
None value is not displayed
```
None Indicates that Nothing is Returned

The special value None represents nothing in Python.

A function that does not explicitly return a value will return None.

*Careful: None is not displayed* by the interpreter as the value of an expression.

```python
def does_not_return_square(x):
    ...  
    x * x  
    ...
>>> does_not_return_square(4)
```

```
>>> sixteen = does_not_return_square(4)
```

None value is not displayed.
None Indicates that Nothing is Returned

The special value `None` represents nothing in Python.

A function that does not explicitly return a value will return `None`.

*Careful:* `None` is *not displayed* by the interpreter as the value of an expression.

```python
>>> def does_not_return_square(x):
...     x * x
... >>> does_not_return_square(4)
```

The name `sixteen` is now bound to the value `None`.

```python
>>> sixteen = does_not_return_square(4)
```
None Indicates that Nothing is Returned

The special value `None` represents nothing in Python.

A function that does not explicitly return a value will return `None`.

*Careful:* `None` is *not displayed* by the interpreter as the value of an expression.

```python
def does_not_return_square(x):
    ...  # x * x
    ...

>>> does_not_return_square(4)

No return

The name `sixteen` is now bound to the value `None`

```python
>>> sixteen = does_not_return_square(4)

>> sixen + 4

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'
Pure Functions & Non-Pure Functions

**Pure Functions**
*just return values*

**Non-Pure Functions**
*have side effects*
Pure Functions & Non-Pure Functions

**Pure Functions**

*just return values*

```python
abs
```

**Non-Pure Functions**

*have side effects*
Pure Functions & Non-Pure Functions

Pure Functions
just return values

Non-Pure Functions
have side effects
Pure Functions & Non-Pure Functions

Pure Functions
just return values

-2 \rightarrow \text{abs} \rightarrow 2

Non-Pure Functions
have side effects
Pure Functions & Non-Pure Functions

**Pure Functions**
*just return values*

-2 ➔ \text{abs} ➔ 2

**Non-Pure Functions**
*have side effects*
Pure Functions & Non-Pure Functions

**Pure Functions**

*just return values*

-2 → \text{abs} → \text{2}

**Argument** → \text{Return value}

**Non-Pure Functions**

*have side effects*
Pure Functions & Non-Pure Functions

**Pure Functions**
just return values

-2 ➔ abs ➔ Return value 2

-2 ➔ pow

**Non-Pure Functions**
have side effects
Pure Functions & Non-Pure Functions

**Pure Functions**
just return values

**Argument**
-2 ➔ abs ➔ 2

Non-Pure Functions
have side effects

2, 100 ➔ pow ➔
Pure Functions & Non-Pure Functions

**Pure Functions**
*just return values*

```
Argument
-2 ➔ abs ➔ 2
```

```
Argument
2, 100 ➔ pow ➔ 2
```

**Non-Pure Functions**
*have side effects*
Pure Functions & Non-Pure Functions

Pure Functions
just return values

-2 ➔ abs ➔ 2

Argument

2, 100 ➔ pow ➔ 126765060022829401496703205376

Return value

2 Arguments

Non-Pure Functions
have side effects
Pure Functions & Non-Pure Functions

Pure Functions
just return values

-2 ➔ abs ➔ 2

2, 100 ➔ pow ➔ 126765060022829401496703205376

Non-Pure Functions
have side effects

print
Pure Functions & Non-Pure Functions

Pure Functions
*just return values*

-2 ➤ abs ➤ 2

2, 100 ➤ pow ➤ 1267650600228229401496703205376

Non-Pure Functions
*have side effects*

-2 ➤ print
Pure Functions & Non-Pure Functions

**Pure Functions**
*just return values*

-2 ➔ abs ➔ 2

-2 ➔ print ➔ None

2, 100 ➔ pow ➔ 1267650600228229401496703205376

**Non-Pure Functions**
*have side effects*

2 ➔ 2 Arguments

None
Pure Functions & Non-Pure Functions

**Pure Functions**
*just return values*

-2 ➔ `abs` ➔ 2

2, 100 ➔ `pow` ➔ 1267650600228229401496703205376

**Non-Pure Functions**
*have side effects*

-2 ➔ `print` ➔ None

*Python displays the output “−2”*
Pure Functions & Non-Pure Functions

**Pure Functions**
*just return values*

-2 ➔ abs ➔ Return value 2

2, 100 ➔ pow ➔ 1267650600228229401496703205376

**Non-Pure Functions**
*have side effects*

-2 ➔ print ➔ Returns None!

*Python displays the output “-2”*
Pure Functions & Non-Pure Functions

**Pure Functions**
*just return values*

-2 ➔ abs ➔ 2

**Argument**

2, 100 ➔ pow ➔ 1267650600228229401496703205376

**2 Arguments**

**Return value**

**Non-Pure Functions**
*have side effects*

-2 ➔ print ➔ None

**Python displays the output “−2”**

A side effect isn't a value; it's anything that happens as a consequence of calling a function
Pure Functions & Non-Pure Functions

**Pure Functions**
*just return values*

- Argument `-2` passed to `abs` function
  - Return value `2`

- Argument `2, 100` passed to `pow` function
  - Return value `1267650600228229401496703205376`

**Non-Pure Functions**
*have side effects*

- Argument `-2` passed to `print` function
  - Returns `None`
  - Python displays the output “-2”

A side effect isn't a value; it's anything that happens as a consequence of calling a function.

(Demo)
Nested Expressions with Print

```python
>>> print(print(1), print(2))
1
2
None None
```
Nested Expressions with Print

```python
>>> print(print(1), print(2))
1
2
None None
```

```python
print(print(1), print(2))
```
Nested Expressions with Print

```python
>>> print(print(1), print(2))
1
2
None None
```

```
print(print(1), print(2))
```
Nested Expressions with Print

```python
>>> print(print(1), print(2))
1
2
None None
```

```python
func print(...)

print(print(1), print(2))
```
Nested Expressions with Print

```python
>>> print(print(1), print(2))
1
2
```

```
func print(...)

print(print(1), print(2))

func print(...) 1
```
Nested Expressions with Print

```python
>>> print(print(1), print(2))
1
2
None None
```
Nested Expressions with Print

```python
>>> print(print(1), print(2))
1
2
None None
```
Nested Expressions with Print

>>> print(print(1), print(2))
1
2
None None
Nested Expressions with Print

```python
>>> print(print(1), print(2))
1
2
None None
```
Nested Expressions with Print

```
>>> print(print(1), print(2))
1
2
None None
```
Nested Expressions with Print

```
>>> print(print(1), print(2))
1
2
None None
```
Nested Expressions with Print

```python
>>> print(print(1), print(2))
1
2
None None
```

```
None, None  ➤ print(...):
            ➤ None
            ➤ display "None None"

None  ➤ print(print(1), print(2))

None
    ➤ print(1)

None
    ➤ print(2)

1  ➤ print(...):
       ➤ None
       ➤ display "1"

2  ➤ print(...):
       ➤ None
       ➤ display "2"
```
Nested Expressions with Print

```
>>> print(print(1), print(2))
1
2
None None
```
Nested Expressions with Print

None, None

print(...):

None

display “None None”

>>> print(print(1), print(2))
1
2
None None

Does not get displayed

None

print(print(1), print(2))

None

print(1)

func print(...)

1

display “1”

None

print(2)

func print(...)

2

display “2”

None

None

print(print(1), print(2))

None

print(1)

func print(...)

1

display “1”

None

print(2)

func print(...)

2

display “2”

None
Multiple Environments
Life Cycle of a User-Defined Function

Def statement:

Call expression:

Calling/Applying:
Life Cycle of a User-Defined Function

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

Call expression:

Calling/Applying:
Life Cycle of a User-Defined Function

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

Call expression:

Calling/Applying:
Life Cycle of a User-Defined Function

Def statement:

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

Call expression:

Calling/Applying:
Life Cycle of a User-Defined Function

Def statement:

Call expression:

Calling/Applying:
Life Cycle of a User-Defined Function

Def statement:

Call expression:

Calling/Applying:
Life Cycle of a User-Defined Function

**Def statement:**

- **Name:**
- **Formal parameter:**
- **Body (return statement):**

**What happens?**

**Call expression:**

**Calling/Applying:**
Life Cycle of a User-Defined Function

**Def statement:**

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

**Call expression:**

```
square(x)
```

**Calling/Applying:**

What happens?
Life Cycle of a User-Defined Function

**Def statement:**

- **Name**: `square(x):`
- **Formal parameter**: `x`
- **Body**: `return mul(x, x)`

**What happens?**

A new function is created!

**Call expression:**

**Calling/Applying:**
Life Cycle of a User-Defined Function

**Def statement:**

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

**Calling/Applying:**

```
>>> def
```

What happens?

- A new function is created!
- Name bound to that function in the current frame

**Call expression:**
Life Cycle of a User-Defined Function

Def statement:

\[
\text{square}(x): \quad \text{return mul}(x, x)
\]

Call expression:

\[
\text{square}(2+2)
\]

What happens?

A new function is created!

Name bound to that function in the current frame

Calling/Applying:
Life Cycle of a User-Defined Function

Def statement:

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

Calling/Applying:

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

What happens?

A new function is created!
Name bound to that function in the current frame
Life Cycle of a User-Defined Function

Def statement:

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

Calling/Applying:

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

What happens?

A new function is created!
Name bound to that function in the current frame
Life Cycle of a User-Defined Function

**Def statement:**
- Name
- Formal parameter
- Body (return statement)
  - Return expression
  - \texttt{return mul(x, x)}

**Call expression:**
- Operator: square
- Function: func \texttt{square(x)}
- Operand: 2+2
  - Argument: 4

**Calling/Applying:**
- \texttt{square(2+2)}

**What happens?**
- A new function is created!
- Name bound to that function in the current frame
- Operator & operands evaluated
Life Cycle of a User-Defined Function

Def statement: 
- Name: `square(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: 
- A new function is created! 
- Name bound to that function in the current frame
- Operator & operands evaluated 
- Function (value of operator) called on arguments (values of operands)
Life Cycle of a User-Defined Function

Def statement:

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

Calling/Applying:
square(2+2)

What happens?

A new function is created!
Name bound to that function in the current frame

Operator & operands evaluated
Function (value of operator) called on arguments (values of operands)
Life Cycle of a User-Defined Function

Def statement:  
`square(x):`

- Name: `square`
- Formal parameter: `x`
- Body (return statement): `return mul(x, x)`

What happens?
- A new function is created!
- Name bound to that function in the current frame

Call expression:  
`square(2+2)`

- Operator: `square`
- Function: `func square(x)`
- Signature

Calling/Applying:

- Operand: `2+2`
- Argument: `4`
- Operator & operands evaluated
- Function (value of operator) called on arguments (values of operands)
Life Cycle of a User-Defined Function

Def statement:

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

What happens?

A new function is created!

Name bound to that function in the current frame

Call expression:

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

Operator & operands evaluated

Function (value of operator) called on arguments (values of operands)

Calling/Applying:

```
square(x):
```

Signature
Life Cycle of a User-Defined Function

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

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

**What happens?**
- A new function is created!
- Name bound to that function in the current frame

Operator & operands evaluated
- Function (value of operator) called on arguments (values of operands)
Life Cycle of a User-Defined Function

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

What happens?
- A new function is created!
  - Name bound to that function in the current frame

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

Calling/Applying: 
- Argument: `4`
  - Signature: `16`
Life Cycle of a User-Defined Function

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

**Call expression:**
- Operator: `square(2+2)`
- Argument: `4` (value of operand `2+2`)
- Function (value of operator) called on arguments
- Signature: `func square(x)`

**Calling/Applying:**
- Argument: `4`
- Signature: `square(x):`
- Return value: `16`

**What happens?**
- A new function is created!
- Name bound to that function in the current frame
- Operator & operands evaluated
- Function (value of operator) called on arguments (values of operands)
Life Cycle of a User-Defined Function

Def statement: `def square(x):
    return mul(x, x)`

Calling/Applying: `square(2+2)`

What happens?

A new function is created!
Name bound to that function in the current frame

Operator & operands evaluated
Function (value of operator) called on arguments (values of operands)

A new frame is created!
Life Cycle of a User-Defined Function

Def statement: `def square(x): return mul(x, x)`

What happens?
- A new function is created!
- Name bound to that function in the current frame

Call expression: `square(2+2)`

What happens?
- Operator & operands evaluated
- Function (value of operator) called on arguments (values of operands)

Calling/Applying: Argument `4` to `square(x): return mul(x, x)`

What happens?
- A new frame is created!
- Parameters bound to arguments
- Return value `16`
Life Cycle of a User-Defined Function

Def statement:  
```
def square(x):
    return mul(x, x)
```

Calling/Applying:  
```
square(2+2)
```

What happens?

A new function is created!
Name bound to that function in the current frame

Operator & operands evaluated
Function (value of operator) called on arguments (values of operands)

A new frame is created!
Parameters bound to arguments
Body is executed in that new environment
Multiple Environments in One Diagram!

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

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

**Interactive Diagram**

![Interactive Diagram]
Multiple Environments in One Diagram!

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

Interactive Diagram
Multiple Environments in One Diagram!

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

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

Interactive Diagram
Multiple Environments in One Diagram!

```python
from operator import mul

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

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

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

```python
from operator import mul

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

square(square(3))
```

Interactive Diagram
Multiple Environments in One Diagram!

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

square(square(3))
```

Interactive Diagram
Multiple Environments in One Diagram!

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

```python
from operator import mul

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

square(square(3))
```

Interactive Diagram
Multiple Environments in One Diagram!

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

Interactive Diagram
Multiple Environments in One Diagram!

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

Interactive Diagram
Multiple Environments in One Diagram!

```python
from operator import mul

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

square(square(3))
```

Interactive Diagram
Multiple Environments in One Diagram!

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

An environment is a sequence of frames.
Multiple Environments in One Diagram!

An environment is a sequence of frames.

- The global frame alone
- A local, then the global frame

Interactive Diagram
Multiple Environments in One Diagram!

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

An environment is a sequence of frames.

- The global frame alone
- A local, then the global frame

Interactive Diagram
Multiple Environments in One Diagram!

An environment is a sequence of frames.

- The global frame alone
- A local, then the global frame

Interactive Diagram
Multiple Environments in One Diagram!

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

An environment is a sequence of frames.
- The global frame alone
- A local, then the global frame

Interactive Diagram
Names Have No Meaning Without Environments

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

An environment is a sequence of frames.
- The global frame alone
- A local, then the global frame

Interactive Diagram
Names Have No Meaning Without Environments

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

An environment is a sequence of frames.

- The global frame alone
- A local, then the global frame

Interactive Diagram
Names Have No Meaning Without Environments

Every expression is evaluated in the context of an environment.

A name evaluates to the value bound to that name in the earliest frame of the current environment in which that name is found.

An environment is a sequence of frames.

- The global frame alone
- A local, then the global frame

Interactive Diagram
Names Have No Meaning Without Environments

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

Every expression is evaluated in the context of an environment.

A name evaluates to the value bound to that name in the earliest frame of the current environment in which that name is found.

An environment is a sequence of frames.

- The global frame alone
- A local, then the global frame

Interactive Diagram
Names Have No Meaning Without Environments

Every expression is evaluated in the context of an environment.

A name evaluates to the value bound to that name in the earliest frame of the current environment in which that name is found.

An environment is a sequence of frames.

- The global frame alone
- A local, then the global frame

Interactive Diagram
Names Have No Meaning Without Environments

Every expression is evaluated in the context of an environment.

A name evaluates to the value bound to that name in the earliest frame of the current environment in which that name is found.

An environment is a sequence of frames.

- The global frame alone
- A local, then the global frame

Interactive Diagram
Names Have Different Meanings in Different Environments

Every expression is evaluated in the context of an environment.

A name evaluates to the value bound to that name in the earliest frame of the current environment in which that name is found.
Names Have Different Meanings in Different Environments

A call expression and the body of the function being called are evaluated in different environments.

Every expression is evaluated in the context of an environment.

A name evaluates to the value bound to that name in the earliest frame of the current environment in which that name is found.
A call expression and the body of the function being called are evaluated in different environments.

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

Every expression is evaluated in the context of an environment.

A name evaluates to the value bound to that name in the earliest frame of the current environment in which that name is found.
Names Have Different Meanings in Different Environments

A call expression and the body of the function being called are evaluated in different environments.

Every expression is evaluated in the context of an environment.

A name evaluates to the value bound to that name in the earliest frame of the current environment in which that name is found.

Interactive Diagram
Names Have Different Meanings in Different Environments

A call expression and the body of the function being called are evaluated in different environments.

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

Every expression is evaluated in the context of an environment.

A name evaluates to the value bound to that name in the earliest frame of the current environment in which that name is found.
Names Have Different Meanings in Different Environments

A call expression and the body of the function being called are evaluated in different environments.

Every expression is evaluated in the context of an environment.

A name evaluates to the value bound to that name in the earliest frame of the current environment in which that name is found.
Miscellaneous Python Features

Division
Multiple Return Values
Source Files
Doctests
Default Arguments

(Demo)
Conditional Statements
A statement is executed by the interpreter to perform an action


Statements

A statement is executed by the interpreter to perform an action

Compound statements:

<header>:
  <statement>
  <statement>
  ...
<separating header>:
  <statement>
  <statement>
  ...
  ...
  ...
  ...
A statement is executed by the interpreter to perform an action.

**Compound statements:**

```
(header>:
  <statement>
  <statement>
  <statement>
...
<separating header>:
  <statement>
  <statement>
  <statement>
...
...
```
A statement is executed by the interpreter to perform an action.

**Compound statements:**

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

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

**Compound statements:**

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

<separating header>:</n
  <statement>
  <statement>
  <statement>
  ...
  ...
```
A *statement* is executed by the interpreter to perform an action.

**Compound statements:**

The first header determines a statement's type.
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
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:

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

Compound statements:

A suite is a sequence of statements

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>`
  - `<statement>`
  - ...
  - Suite

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

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

- Execute the first statement
- Unless directed otherwise, execute the rest

A suite is a sequence of statements

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

(Demo)
Conditional Statements

(Demo)

```python
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

(Demo)

```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
Conditional Statements

(Demo)

```python
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

(Demo)

```python
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 remaining clauses.
Conditional Statements

(Demo)

```python
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 remaining clauses.

Syntax Tips:
Conditional Statements

(Demo)

```python
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 remaining clauses.

Syntax Tips:

1. Always starts with "if" clause.
2. Zero or more "elif" clauses.
3. Zero or one "else" clause, always at the end.
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
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
```

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

Boolean Contexts

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

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

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

True values in Python:  Anything else (True)

Read Section 1.5.4!

Reading: http://composingprograms.com/pages/15-control.html#conditional-statements
Iteration
While Statements

(Demo)
While Statements

(Demo)

```
1 i, total = 0, 0
2 while i < 3:
3     i = i + 1
4     total = total + i
```
While Statements

(Demo)

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

```
1  i, total = 0, 0
2  while i < 3:
3    i = i + 1
4    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.
While Statements

(Demo)

1 \( i, \) total = 0, 0
2 while \( i < 3 \):
3 \( i = i + 1 \)
4 \( \text{total} = \text{total} + i \)

**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.
**While Statements**

(Demo)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>i, total = 0, 0</td>
</tr>
<tr>
<td>2</td>
<td>while <strong>i &lt; 3</strong>:</td>
</tr>
<tr>
<td>3</td>
<td>i = i + 1</td>
</tr>
<tr>
<td>4</td>
<td>total = total + i</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.
While Statements

(Demo)

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

```plaintext
1  i, total = 0, 0
2  while i < 3:
3      i = i + 1
4      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.
While Statements

(Demo)

1. Evaluate the header’s expression.

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

Global frame

| i | 0 |
| total | 0 |

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.
While Statements

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.

(Demo)

1  i, total = 0, 0
2  while i < 3:
3      i = i + 1
4      total = total + i

Global frame
   i  0
   total  0
While Statements

(Demo)

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.
While Statements

(Demo)

1  i, total = 0, 0
2  while i < 3:
   3     i = i + 1
   4     total = total + i

Global frame
   i 1
      total 0

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.
While Statements

(Demo)

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.
While Statements

(Demo)

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

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

Global frame

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>1</td>
</tr>
<tr>
<td>total</td>
<td>1</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.
While Statements

(Demo)

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.

```
1 i, total = 0, 0
2 while i < 3:
3   i = i + 1
4   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.
While Statements

(Demo)

1  i, total = 0, 0
2  while i < 3:
3    i = i + 1
4    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.
While Statements

(Demo)

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

```
Global frame
    i | x x 2
    total | x 1

1  i, total = 0, 0
2  while [i < 3]:
3     i = i + 1
4     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.
While Statements

(Demo)

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

```
1 i, total = 0, 0
2 while [i < 3]:
3   i = i + 1
4   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.
While Statements

(Demo)

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

<table>
<thead>
<tr>
<th>i</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Global frame

<table>
<thead>
<tr>
<th>i</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>total</td>
<td>3</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.
While Statements

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.

(Demo)

1. i, total = 0, 0
2. while i < 3:
3. i = i + 1
4. total = total + i

Global frame

<table>
<thead>
<tr>
<th>i</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>total</td>
<td>3</td>
</tr>
</tbody>
</table>
While Statements

(Demo)

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

<table>
<thead>
<tr>
<th>i, total = 0, 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>while i &lt; 3:</td>
</tr>
<tr>
<td>3 i = i + 1</td>
</tr>
<tr>
<td>4 total = total + i</td>
</tr>
</tbody>
</table>

Global frame

<table>
<thead>
<tr>
<th>i</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>total</td>
<td>3</td>
<td></td>
<td></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.
While Statements

(Demo)

1 i, total = 0, 0
2 while $i < 3$:
3 i = i + 1
4 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.
While Statements

(Demo)

1  i, total = 0, 0
2  while i < 3:
3      i = i + 1
4  total = total + i

Global frame

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


While Statements

(Demo)

1 \(i, \text{total} = 0, 0\)
2 \(\text{while } i < 3:\)
3 \(i = i + 1\)
4 \(\text{total} = \text{total} + i\)

Global frame
\[
\begin{array}{c}
i \quad 3 \\
\text{total} \quad 6
\end{array}
\]

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.