Control
Announcements
Print and None

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

The special value `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
    ...

>>> does_not_return_square(4)
```

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 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)
```

Note: The value is not displayed, hence the comment "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)
16
>>> sixteen = does_not_return_square(4)
None value is not displayed
```

The name `sixteen` is now bound to the value `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

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

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

**Pure Functions**

*just return values*

-2 \(\rightarrow\) `abs`

**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

![Diagram showing -2 passed to abs function, which returns 2]

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

**Pure Functions**  
just return values

**Argument**  
\(-2\)  
\[
\text{abs}
\]

**Return value**  
\(2\)

**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*

  - Argument: `-2`
  - Function: `abs`
  - Return value: `2`

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

  - Argument: `2, 100`
  - Function: `pow`
Pure Functions & Non-Pure Functions

**Pure Functions**
*just return values*

-2 → **abs** → 2

**Argument**

2, 100 → **pow** → **Return value**

**2 Arguments**

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

**Pure Functions**

*just return values*

- Argument: $-2$
  - Function: abs
  - Return value: 2

- Argument: $2, 100$
  - Function: pow
  - Return value: $126765060022829401496703205376$

**Non-Pure Functions**

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

**Pure Functions**
just return values

- **Argument**
  - `-2` → `abs` → **Return value**
    - `2`
  - `2, 100` → `pow` →
    - **Return value**
      - `1267650600228229401496703205376`

- **2 Arguments**

**Non-Pure Functions**
have side effects

- **print**
Pure Functions & Non-Pure Functions

**Pure Functions**
*just return values*

-2 ➔ `abs` ➔ Return value ➔ 2

2, 100 ➔ `pow` ➔ Return value ➔ 1267650600228229401496703205376

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

-2 ➔ `print`
Pure Functions & Non-Pure Functions

Pure Functions
- just return values

-2 \( \rightarrow \) abs \( \rightarrow 2 \)

2, 100 \( \rightarrow \) pow \( \rightarrow 126765060022829401496703205376 \)

2 Arguments

Non-Pure Functions
- have side effects

-2 \( \rightarrow \) print \( \rightarrow \) None
Pure Functions & Non-Pure Functions

**Pure Functions**
just return values

- Argument
  - -2
  - abs
  - 2

- Argument
  - 2, 100
  - pow
  - Return value
  - 126765060022829401496703205376

**Non-Pure Functions**
have side effects

- Argument
  - -2
  - print
  - Return value
  - None

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

**Pure Functions**
just return values

-2 ➔ `abs`
  - Argument
  - Return value ➔ 2

2, 100 ➔ `pow`
  - 2 Arguments
  - Returns None!
  - 126765060022829401496703205376

**Non-Pure Functions**
have side effects

-2 ➔ `print`
  - Python displays the output “-2”
  - 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

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

Returns None!
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
```

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

```python
func print(...) {
    print(print(1), print(2))
}

>>> 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

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

```
def print(...):
    display "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

```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

```python
none, none  # print(...):
           None

display "None None"

>>> 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

None, None → `print(...):` → None

display “None None”

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

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

display “None None”
Multiple Environments
Life Cycle of a User-Defined Function

Def statement:

Call expression:

Calling/Applying:

What happens?
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)

What happens?

Call expression:

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

Def statement:

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

What happens?

Call expression:

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

Def statement:

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

What happens?

Call expression:

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

Def statement: 

Formal parameter

Name

square(x):

return mul(x, x)

What happens?

Def statement

Call expression:

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

Def statement:

Def statement

Name

Formal parameter

square(x):

Body (return statement)

return mul(x, x)

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(8)
```

**Calling/Applying:**

What happens?
Life Cycle of a User-Defined Function

**Def statement:**

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

What happens?

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

Call expression:

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

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

Call expression:  
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:**
```
def square(x):
    return mul(x, x)
```

**Call expression:**
```
square(2+2)
```

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

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

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

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

**Calling/Applying:**
- square(2+2)
- Name bound to that function in the current frame
Life Cycle of a User-Defined Function

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

Calling/Applying:
- `square(2+2)`
  - 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:**
- 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:**
- `square(x):`

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

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:

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

What happens?

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

Call expression:

- **Operator**: `square(2+2)`
- **Function**: `func square(x)`
- **Signature**: 4
- **Operand**: 2+2
  - **Argument**: 4

Calling/Applying:

- **Value**: 4
Life Cycle of a User-Defined Function

Def statement:
- **Name**: square(x):
  - **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:
- **Operator & operands evaluated**
- **Function (value of operator) called on arguments (values of operands)**

Calling/Applying:
- **Signature**: 4
- **Operand**: 2+2
  - **Argument**: 4
  - **Operator**: square
    - **Function**: `func square(x)`
  - **Value**: 16
Life Cycle of a User-Defined Function

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

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

**Calling/Applying:**
- Argument: `4`
- Signature: `mul(x, x)` (evaluated to `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:

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

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

What happens?

Call expression:

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

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

Calling/Applying:

- **Argument**: 4
- **Signature**: square(x)
- **Return value**: 16

Argument
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!

Operand: 2+2  Argument: 4

Operator: square  function: func square(x)

Signature

Argument

Return value

8
Life Cycle of a User-Defined Function

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

Calling/Applying:
- Operator: square
- Function: func square(x)

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
Life Cycle of a User-Defined Function

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

**Calling/Applying:**
- **Operator & operands evaluated:**
  - **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)
- 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
from operator import mul

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

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

```python
from operator import mul

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

square(square(3))
```

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

```python
from operator import mul

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

square(square(3))
```

---

**Global frame**

- `func mul(...)`
- `func square(x) [parent=Global]`
Multiple Environments in One Diagram!

```python
from operator import mul

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

square(square(3))
```

Global frame

- `func mul(...)`
- `func square(x) [parent=Global]`

- `square(square(3))`
- `func square(x)`
Multiple Environments in One Diagram!

```python
from operator import mul

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

square(square(3))
```

[Python code diagram]
Multiple Environments in One Diagram!

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

Global frame

```
func mul(...)  
mul

func square(x) [parent=Global]
```

```
square(square(3))

func square(x)

square(3)
```

```
func square(x)
```
Multiple Environments in One Diagram!

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

```python
from operator import mul

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

square(square(3))
```

![Diagram showing multiple environments in one diagram](http://pythontutor.com/composingprograms.html?code=from%20operator%20import%20mul%0Adef%20square(x)%3A%0A%20%20%20%20return%20mul(x,%20x)%0A%0Ar%20%20%20%20square(square(3)))&cumulative=true&curInstr=0&mode=display&origin=composingprograms.js&py=3&rawInputLstJSON=%5B%5D)
Multiple Environments in One Diagram!

```python
from operator import mul

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

square(square(3))
```

![Diagram showing multiple environments in one diagram](https://pythontutor.com/composingprograms/tree?_preview=from%20operator%20import%20mul%0Adef%20square(x)%3A%0A%20%20%20%20return%20mul(x,%20x)%0Asquare(square(3)))&cumulative=true&curInstr=0&mode=display&origin=composingprograms.js&py=3&rawInputLstJSON=%5B%5D)
Multiple Environments in One Diagram!

```python
from operator import mul

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

square(square(3))
```

Global frame

```
func mul(...)
mul
square
func square(x) [parent=Global]
```

```
f1: square [parent=Global]
x    3
    Return value
    9
```

Diagram showing the nested function calls and the return value.
Multiple Environments in One Diagram!

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

```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))

```
func square(x) [parent=Global]
```

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

```
1: square [parent=Global]
x  3

Return value 9
```

```
http://pythontutor.com/composingprograms.html#code=from%20operator%20import%20mul%0Adef%20square%28x%29%3A%0A%20%20%20%20r ... Asquare%28square%283%29%29&cumulative=true&curInstr=0&mode=display&origin=composingprograms.js&py=3&rawInputLstJSON=%5B%5D
```
Multiple Environments in One Diagram!

```python
from operator import mul

def square(x):
    return mul(x, x)
square(square(3))
```

Diagram showing the computation of `square(square(3))` with multiple environments and function calls.
Multiple Environments in One Diagram!

An environment is a sequence of frames.

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

```
Global frame

func mul(…)

mul

func square(x) [parent=Global]

square

f1: square [parent=Global]

x 3

Return value 9

f2: square [parent=Global]

x 9

Return value 81

81

square(square(3))

func square(x)

9

square(3)

func square(x)

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
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.
- The global frame alone
- A local, then the global frame
Multiple Environments in One Diagram!

```python
from operator import mul

def square(x):
    return mul(x, x)
square(square(3))
```

An environment is a sequence of frames.

- The global frame alone
- A local, then the global frame
Multiple Environments in One Diagram!

An environment is a sequence of frames.

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

```python
from operator import mul

def square(x):
    return mul(x, x)
square(square(3))
```
Names Have No Meaning Without Environments

An environment is a sequence of frames.

- The global frame alone
- A local, then the global frame
Names Have No Meaning Without Environments

Every expression is evaluated in the context of an environment.

An environment is a sequence of frames.

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

http://pythontutor.com/composingprograms.html?&code=from operator import mul
  def square(x):
      return mul(x, x)
  square(square(3))
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
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
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
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

http://pythontutor.com/composingprograms.html#code=from%20operator%20import%20mul%0Adef%20square(x)%3A%0A%20%20%20%20r...Asquare%28square%283%29%29&cumulative=true&curInstr=0&mode=display&origin=composingprograms.js&py=3&rawInputLstJSON=%5B%5D
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.
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.
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.

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

Global frame

```
f1: square [parent=Global]
```

```
square 4
Return value 16
```
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.

```
from operator import mul

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

square(4)
```

HTTP://pythontutor.com/composingprograms.html?htmlCode=from%20operator%20import%20mul%0Adef%20square(square%29%3A%0A%20%20%20%20%20return%20mul(square,*square)%29%0Asquare(4)&cumulative=true&curInstr=0&mode=display&origin=composingprograms.js&py=3&rawInputLstJSON=%5B%5D
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.
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:

Statement

Clause

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

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

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

**Compound statements:**

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

<separating header>
  <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:**

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

A suite is a sequence of statements
Compound Statements

**Compound statements:**

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

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

• Unless directed otherwise, execute the rest
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
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:
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

```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:**

1. Evaluate the header's expression.

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

**Syntax Tips:**
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
```

Execution Rule for Conditional Statements:

1. Each clause is considered in order.
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
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
```

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

Boolean Contexts

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)

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

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
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)

George Boole

Read Section 1.5.4!

(Demo)

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

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)

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

```python
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

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
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

```
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

(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.

```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.

Global frame

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

```
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.

```
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.

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

Global frame

<table>
<thead>
<tr>
<th>i</th>
<th>1</th>
</tr>
</thead>
<tbody>
<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 i, total = 0, 0
2 while i < 3:
3 \[ \text{i = i + 1} \]
4 \[ \text{total = total + i} \]

Global frame

<table>
<thead>
<tr>
<th>i</th>
<th>(\times) 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>total</td>
<td>(\times) 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.
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

```
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.
### 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

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

(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 i &lt; 3:</td>
</tr>
<tr>
<td>3</td>
<td>i = i + 1</td>
</tr>
<tr>
<td>4</td>
<td>total = total + i</td>
</tr>
</tbody>
</table>

Global frame

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>2</td>
</tr>
<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.
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

George Boole

(Demo)

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>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td></td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>total</td>
<td></td>
<td></td>
<td></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.
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

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   x  x  x  3
total x  x  x  6
```
Example: Prime Factorization
Prime Factorization
Prime Factorization

Each positive integer $n$ has a set of prime factors: primes whose product is $n$. 
Prime Factorization

Each positive integer $n$ has a set of prime factors: primes whose product is $n$

... 
8  = 2 * 2 * 2 
9  = 3 * 3 
10 = 2 * 5 
11 = 11 
12 = 2 * 2 * 3 
...

Prime Factorization

Each positive integer $n$ has a set of prime factors: primes whose product is $n$

...  
8 $= 2 \times 2 \times 2$
9 $= 3 \times 3$
10 $= 2 \times 5$
11 $= 11$
12 $= 2 \times 2 \times 3$
...

One approach: Find the smallest prime factor of $n$, then divide by it
Prime Factorization

Each positive integer n has a set of prime factors: primes whose product is n

...  
8  = 2 * 2 * 2  
9  = 3 * 3  
10 = 2 * 5  
11 = 11  
12 = 2 * 2 * 3  
...  

One approach: Find the smallest prime factor of n, then divide by it

858
Prime Factorization

Each positive integer $n$ has a set of prime factors: primes whose product is $n$

...  
$8 = 2 \times 2 \times 2$  
$9 = 3 \times 3$  
$10 = 2 \times 5$  
$11 = 11$  
$12 = 2 \times 2 \times 3$  
...

One approach: Find the smallest prime factor of $n$, then divide by it

$858 = 2 \times 429$
Prime Factorization

Each positive integer $n$ has a set of prime factors: primes whose product is $n$

$$
\ldots
8 \ = \ 2 \times 2 \times 2 \\
9 \ = \ 3 \times 3 \\
10 \ = \ 2 \times 5 \\
11 \ = \ 11 \\
12 \ = \ 2 \times 2 \times 3 \\
\ldots
$$

One approach: Find the smallest prime factor of $n$, then divide by it

$$
858 \ = \ 2 \times 429 \ = \ 2 \times 3 \times 143
$$
Prime Factorization

Each positive integer $n$ has a set of prime factors: primes whose product is $n$

...  
8 = 2 * 2 * 2  
9 = 3 * 3  
10 = 2 * 5  
11 = 11  
12 = 2 * 2 * 3  
...

One approach: Find the smallest prime factor of $n$, then divide by it

858 = 2 * 429 = 2 * 3 * 143 = 2 * 3 * 11 * 13
Prime Factorization

Each positive integer \( n \) has a set of prime factors: primes whose product is \( n \)

\[
\begin{align*}
8 & = 2 \times 2 \times 2 \\
9 & = 3 \times 3 \\
10 & = 2 \times 5 \\
11 & = 11 \\
12 & = 2 \times 2 \times 3 \\
\end{align*}
\]

One approach: Find the smallest prime factor of \( n \), then divide by it

\[
858 = 2 \times 429 = 2 \times 3 \times 143 = 2 \times 3 \times 11 \times 13
\]

(Demo)