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`.
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 \texttt{None} represents nothing in Python

A function that does not explicitly return a value will return \texttt{None}

\textit{Careful}: \texttt{None} is \emph{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
    ...
>>> 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)
None value is not displayed

>>> 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)
16

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

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)
None

>>> 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'
```

The name `sixteen` is now bound to the value `None`. None value is not displayed.
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 ➔ abs ➔ 2
```

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

**Pure Functions**
*just return values*

![](image)

**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 ➔ abs ➔ 2

**Argument**

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

**Pure Functions**
*just return values*

-2 ➔ abs ➔ Return value ➔ 2

Argument

2, 100 ➔ pow

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

**Pure Functions**
*just return values*

-2 -> **abs** -> Return value -> 2

-2 Arguments

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

2, 100 -> **pow**
Pure Functions & Non-Pure Functions

**Pure Functions**
*just return values*

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

-2 \(\rightarrow\) Argument

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

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

2 Arguments
Pure Functions & Non-Pure Functions

Pure Functions
just return values

-2
Argument
abs
2
Return value

2, 100
pow
126765060022829401496703205376

2 Arguments

Non-Pure Functions
have side effects

print
Pure Functions & Non-Pure Functions

**Pure Functions**
*just return values*

-2

**Argument**

-2

abs

Return value

2

**2 Arguments**

2, 100

pow

126765060022829401496703205376

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

-2

print
Pure Functions & Non-Pure Functions

**Pure Functions**
*just return values*

-2 ➔ abs ➔ 2

-2 ➔ pow ➔ 126765060022829401496703205376

2, 100 ➔ pow ➔ 126765060022829401496703205376

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

-2 ➔ print ➔ None
Pure Functions & Non-Pure Functions

**Pure Functions**  
*just return values*

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

- Argument: `2, 100`  
  - Function: `pow`  
  - Return value: `1267650600228229401496703205376`

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

- Argument: `-2`  
  - Function: `print`  
  - Return value: `None`

*Python displays the output “-2”*
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 ➔ 2

2, 100 ➔ pow ➔ 1267650600228229401496703205376

**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
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
```

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

```python
func print(...)
```
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

```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))
```

```
>>> 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"
def print(...):
    display "2"
```
Nested Expressions with Print

None, None ➔ print(...):

1 ➔ print(...): None

display “None None”

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

```
func print(...)

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

display "None None"
```
Nested Expressions with Print

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

```
>>> print(print(1), print(2))
1
2
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:**
```python
def square(x):
    return mul(x, x)
```

**Call expression:**

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

Def statement:

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

Call expression:

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

**Def statement:**

```python
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:

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

Calling/Applying:

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

- Name
- Formal parameter
- Body (return statement)
- Return expression

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:**

```
def 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)
```

**Call expression:**

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

**Calling/Applying:**

A new function is created!

Name bound to that function in the current frame.
Life Cycle of a User-Defined Function

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

**Call expression:**
- *Operand*: `2+2`
- *Argument*: `4`

**Calling/Applying:**

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

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

**Calling/Applying:**
- Operand: 2+2
- Argument: 4

**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:

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

Calling/Applying:

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

**Call expression:**
- Operator: `square(2+2)`
- Argument: `4`

**Calling/Applying:**
- `square(x)`: Signature

**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):
    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:

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

What happens?

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

Call expression:

- Argument: `2+2`
- Operator: `square`
- Function: `func square(x)`

Calling/Applying:

- Signature: `4
- Signature: `16
- Operand: `2+2`
- Argument: `4`
Life Cycle of a User-Defined Function

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

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

Calling/Applying: `4 \rightarrow square(x):` 16
- Argument
- Signature

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)
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
- Formal parameter: `x`
- Body (return statement): `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)`
- operand: `2+2`
- argument: `4`

What happens?

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

Calling/Applying: `square(x):`

- Argument: `4`
- Signature: `mul(x, x)`
- Return value: `16`
Life Cycle of a User-Defined Function

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

**Call expression:**
- **Operator:** square
- **Function:** func square(x)

**Calling/Applying:**
- **Argument:** 2+2
- **Signature:** 4
- **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)
- A new frame is created!
- Parameters bound to arguments
Life Cycle of a User-Defined Function

Def statement:

Def statement

Name

square(x):

Formal parameter

Return expression

Body (return statement)

Operands: 2+2

Argument: 4

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

Body is executed in that new environment
Life Cycle of a User-Defined Function

**Def statement:**
- Name: `square(x):`
- Body: `return mul(x, x)`
- 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)`
- 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`
- A new frame is created!
- Parameters bound to arguments
- Body is executed in that new environment

(Demo)
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))
```

```
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(...)

mul

func square(x) [parent=Global]

square
```
Multiple Environments in One Diagram!

```python
from operator import mul

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

square(square(3))
```

Diagram showing multiple environments and function calls.
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!

```
from operator import mul

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

square(square(3))
```

![Diagram showing multiple environments with a function `square` and its nested calls.](http://pythontutor.com/composingprograms.html?mode=display&curInstr=0&rawInputLstJSON=%5B%5D)
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))
```

Diagram:
- Global frame
- `func mul(...)`
- `func square(x) [parent=Global]`
- `mul`
- `square`
- `f1: square [parent=Global]`
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%20r...&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))
```

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

```
3
^ Return value
```

```
square(3)
```

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

```
mul
square
```

```
func mul(...)
```

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

```
x 3
```

```
9
```

```
square(square(3))
```

```
square(3)
```

```
func square(x)
```

```
3
```

```
func square(x)
```

```
9
```

```
square(3)
```

```
func square(x)
```

```
square(3)
```

```
func square(x)
```

```
9
```

```
square(3)
```

```
func square(x)
```

```
3
```

```
func square(x)
```

```
9
```

```
square(3)
```

```
func square(x)
```

```
3
```

```
func square(x)
```

```
square(3)
```

```
func square(x)
```

```
9
```

```
square(3)
```

```
func square(x)
```

```
3
```

```
func square(x)
```

```
square(3)
```

```
func square(x)
```

```
9
```

```
square(3)
```

```
func square(x)
```

```
3
```

```
func square(x)
```

```
square(3)
```

```
func square(x)
```

```
9
```

```
square(3)
```

```
func square(x)
```

```
3
```

```
func square(x)
```

```
square(3)
```

```
func square(x)
```

```
9
```

```
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]

x 3
Return value 9

```

```
func mul(...)  

mul

Global frame

square

```

```
square(square(3))

func square(x)  

3

square(3)

```

```
func square(x)  

9

```
Multiple Environments in One Diagram!

```
from operator import mul

# Function to calculate square of a number

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

# Square of 3

square(square(3))
```

Diagram:

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

Function `square(x)`:
- `x`: 3
- Return value: 9

Function `square(square(3))`:
- `x`: 9
- Return value: 81
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.
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!

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))
```
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

```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
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
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
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.

http://pythontutor.com/composingprograms.html?lang=python&mode=display&curInstr=0&cumulative=true&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.
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.

```
from operator import mul

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

square(4)
```
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>
    ...
...
```
Statements

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:

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

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

16
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
## Statements

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>
  <statement>
  ...
  ...
```
Compound Statements

**Compound statements:**

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

A suite is a sequence of 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:

A suite is a sequence of statements

Suite

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:
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:

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

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. Evaluate the header's expression.
3. 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.
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
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

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)
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

False values in Python: False, 0, '', None  
(True)

20
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
```

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](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)

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)

```
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.
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.

 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 [blue]i < 3[blue]: 
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.

```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

(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

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

**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.

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></th>
<th>i, total = 0, 0</th>
</tr>
</thead>
<tbody>
<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

<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. 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
```

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
```

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

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

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.

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

<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.
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
```

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)
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 \( n \)

\[
\begin{align*}
\ldots & \quad = 2 \times 2 \times 2 \\
9 & = 3 \times 3 \\
10 & = 2 \times 5 \\
11 & = 11 \\
12 & = 2 \times 2 \times 3 \\
\ldots
\end{align*}
\]
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
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 \( n \)

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

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

\[
858 \quad = \quad 2 \times 429
\]
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

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

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

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

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

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

\[
\begin{align*}
858 &= 2 \times 429 &= 2 \times 3 \times 143 &= 2 \times 3 \times 11 \times 13 \\
(Demo)
\end{align*}
\]