Announcements

• There's plenty of room in live lecture if you want to come (but videos are still better)
Announcements

• There's plenty of room in live lecture if you want to come (but videos are still better)

• Please don't make noise outside of the previous lecture!
Announcements

• There's plenty of room in live lecture if you want to come (but videos are still better)
• Please don't make noise outside of the previous lecture!
• Homework 1 is due next Wednesday 9/10 at 2pm (Changed from original time!)
Announcements

• There's plenty of room in live lecture if you want to come (but videos are still better)
• Please don't make noise outside of the previous lecture!
• Homework 1 is due next Wednesday 9/10 at 2pm *(Changed from original time!)*
  ▪ Homework is graded on effort, but the bar is high – you must make substantial progress
Announcements

• There's plenty of room in live lecture if you want to come (but videos are still better)

• Please don't make noise outside of the previous lecture!

• Homework 1 is due next Wednesday 9/10 at 2pm (Changed from original time!)
  • Homework is graded on effort, but the bar is high — you must make substantial progress
  • Monday homework parties 3pm–4pm in Wozniak Lounge and 6pm–8pm in 2050 VLSB
Announcements

• There's plenty of room in live lecture if you want to come (but videos are still better)

• **Please** don't make noise outside of the previous lecture!

• Homework 1 is due next Wednesday 9/10 at 2pm (**Changed from original time!**)  
  ▪ Homework is graded on effort, but the bar is high — you must make substantial progress
  ▪ Monday homework parties 3pm–4pm in Wozniak Lounge and 6pm–8pm in 2050 VLSB

• Take-home quiz released next Wednesday 9/10 at 3pm, due Thursday 9/12 at 11:59pm
Announcements

• There's plenty of room in live lecture if you want to come (but videos are still better)

• Please don't make noise outside of the previous lecture!

• Homework 1 is due next Wednesday 9/10 at 2pm (*Changed from original time!*)
  ▪ Homework is graded on effort, but the bar is high — you must make substantial progress
  ▪ Monday homework parties 3pm–4pm in Wozniak Lounge and 6pm–8pm in 2050 VLSB

• Take-home quiz released next Wednesday 9/10 at 3pm, due Thursday 9/12 at 11:59pm
  ▪ 3 points, similar in format to homework, but graded for correctness
Announcements

• There's plenty of room in live lecture if you want to come (but videos are still better)

• Please don't make noise outside of the previous lecture!

• Homework 1 is due next Wednesday 9/10 at 2pm (\textit{Changed from original time!})
  • Homework is graded on effort, but the bar is high — you must make substantial progress
  • Monday homework parties 3pm-4pm in Wozniak Lounge and 6pm-8pm in 2050 VLSB

• Take-home quiz released next Wednesday 9/10 at 3pm, due Thursday 9/12 at 11:59pm
  • 3 points, similar in format to homework, but graded for correctness
  • If you score 0/3, you will need to talk to the course staff or be dropped
Announcements

• There's plenty of room in live lecture if you want to come (but videos are still better)

• Please don't make noise outside of the previous lecture!

• Homework 1 is due next Wednesday 9/10 at 2pm (Changed from original time!)
  ▪ Homework is graded on effort, but the bar is high – you must make substantial progress
  ▪ Monday homework parties 3pm–4pm in Wozniak Lounge and 6pm–8pm in 2050 VLSB

• Take-home quiz released next Wednesday 9/10 at 3pm, due Thursday 9/12 at 11:59pm
  ▪ 3 points, similar in format to homework, but graded for correctness
  ▪ If you score 0/3, you will need to talk to the course staff or be dropped
  ▪ Open-computer: You can use the Python interpreter, watch course videos, etc.
Announcements

• There’s plenty of room in live lecture if you want to come (but videos are still better)

• Please don’t make noise outside of the previous lecture!

• Homework 1 is due next Wednesday 9/10 at 2pm (Changed from original time!)
  * Homework is graded on effort, but the bar is high – you must make substantial progress
  * Monday homework parties 3pm–4pm in Wozniak Lounge and 6pm–8pm in 2050 VLSB

• Take-home quiz released next Wednesday 9/10 at 3pm, due Thursday 9/12 at 11:59pm
  * 3 points, similar in format to homework, but graded for correctness
  * If you score 0/3, you will need to talk to the course staff or be dropped
  * Open–computer: You can use the Python interpreter, watch course videos, etc.
  * Closed–help: Please don't talk to your classmates, search for answers, etc.
Announcements

• There's plenty of room in live lecture if you want to come (but videos are still better)
• Please don't make noise outside of the previous lecture!
• Homework 1 is due next Wednesday 9/10 at 2pm (Changed from original time!)
  ▪ Homework is graded on effort, but the bar is high – you must make substantial progress
  ▪ Monday homework parties 3pm–4pm in Wozniak Lounge and 6pm–8pm in 2050 VLSB
• Take-home quiz released next Wednesday 9/10 at 3pm, due Thursday 9/12 at 11:59pm
  ▪ 3 points, similar in format to homework, but graded for correctness
  ▪ If you score 0/3, you will need to talk to the course staff or be dropped
  ▪ Open–computer: You can use the Python interpreter, watch course videos, etc.
  ▪ Closed–help: Please don't talk to your classmates, search for answers, etc.
• Project 1 due Wednesday 9/17 at 11:59pm.
Multiple Environments
Life Cycle of a User-Defined Function

What happens?

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)

What happens?

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:

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

What happens?

Call expression:

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

Def statement:
- Name
- Formal parameter
- Body

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)

Body (return statement)

What happens?

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: 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: square(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:

Calling/Applying:
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

Call expression: 
\[ \text{square}(2+2) \]

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

**Def statement:**
```
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)
```

**Calling/Applying:**
- operator: square
- function: `func square(x)`
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

**Calling/Applying:**
- `operator: square`
- `function: func square(x)`
- `operand: 2+2`
- `argument: 4`
Life Cycle of a User-Defined Function

Def statement:

Formal parameter
Name
return
expression
Body (return statement)

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

Call expression:

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

Operator & operands evaluated

What happens?

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

**Def statement:**
- Name: `square(x):`
- Formally 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: `square(2+2)`
- Operand: `2+2` argument: `4`
- Operator function: `func square(x)`

**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

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

**Calling/Applying:**
- **4**
- **Function (value of operator) called on arguments (values of operands):**
  - **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:

```
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:

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

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

Calling/Applying:

```
Calling/Applying: 4
```

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

```python
>>> square(2+2)
16
```
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
- Formal parameter: \(x\)
- Body (return statement): \(\text{return mul}(x, x)\)

Call expression: 
- Operator: \(\text{square}(2+2)\)
- 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)

Calling/Applying: 
- Argument
- Signature
- Return value: 16
Life Cycle of a User-Defined Function

Def statement:

- **Name**: square(x): 
- **Formal parameter**: 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 & operands evaluated**
- **Function (value of operator) called on arguments (values of operands)**

Calling/Applying:

- **Argument**: 2+2 
- **Signature**: square(x): 
- **Return value**: 4

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

Def statement:

- **square(\(x\))**: return mul(\(x, x\))

Calling/Applying:

- \(\text{square}(2+2)\)

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
**Life Cycle of a User-Defined Function**

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

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

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

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

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

**What happens?**
- A new frame is created!
- Parameters bound to arguments
- Body is executed in that new environment
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
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))
```

`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))
```
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!

```
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
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))
```

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))
```

Interactive 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))
```

An environment is a sequence of frames.

Interactive Diagram
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

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!

An environment is a sequence of frames.

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

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

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
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.
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.

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.

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**

```python
from operator import mul
def square(square):
    return mul(square, square)
square(4)
```
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
Miscellaneous Python Features

Operators
Multiple Return Values
Docstrings
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>
  ...  
  ...
Statements

A statement is executed by the interpreter to perform an action

Compound statements:

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

Compound statements:
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:

The first header determines a statement’s type
**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.
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>`
  - `<statement>`
  - ...

  ![Suite](image)
Compound Statements

Compound statements:

A suite is a sequence of statements

Suite
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

**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

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

(Demo)

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:

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:**

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

Boolean Contexts

George Boole
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

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

Boolean Contexts

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

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)

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

(Demo)

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</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>

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

While Statements

(Demo)

<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></th>
<th>i</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<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.
While Statements

(Demo)

1. $i, \text{total} = 0, 0$
2. \text{while } i < 3: \quad \text{(Global frame)}
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

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

- 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.

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

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 = 2
- total = 1
While Statements

(Demo)

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

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

Global frame
    i  2
    total  3

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>Global frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>i × × × 3</td>
</tr>
<tr>
<td>2</td>
<td>total × × 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)

<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></th>
<th>i</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>total</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>
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.