Announcements
Multiple Environments
Life Cycle of a User-Defined Function

Def statement:

Call expression:

What happens:

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

Def statement:

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

Calling/Applying:

What happens?
Life Cycle of a User-Defined Function

**Def statement:**

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

**Calling/Applying:**

```python
>>> def
```

What happens?
Life Cycle of a User-Defined Function

Def statement:

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

What happens?

Calling/Applying:

```
>>> def
```

Call expression:
Life Cycle of a User-Defined Function

Def statement:

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

What happens?

Calling/Applying:

```
square(4)  # Calling the function with a value of 4
```
Life Cycle of a User-Defined Function

Def statement:

Name -> Formal parameter

Def statement

Body

Calling/Applying:

square(x):

return mul(x, x)

What happens?
Life Cycle of a User-Defined Function

Def statement:

Call expression:

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

Def statement:

Call expression:

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

Def statement:

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

What happens?

A new function is created!

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

**Def statement:**

- **square**
  - Formal parameter: \( x \)
  - Return expression: \( \text{mul}(x, x) \)
  - Body (return statement)

**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:

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

Calling/Applying:

```python
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:
- **square** (x):
  - 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
Life Cycle of a User-Defined Function

Def statement:

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

Calling/Applying:

- **Operator**: `square` (2+2)
- **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:
```
def square(x):
    return mul(x, x)
```

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

Calling/Applying:

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

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

**Def statement:**

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

**Call expression:**

- Operator: `2+2`
- Argument: `4`

**Calling/Applying:**

- **Def statement**: 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).

What happens?

A new function is created!

Name bound to that function in the current frame

Operator & operands evaluated

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

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

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

**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:

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

Call expression:

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

Calling/Applying:

- **Signature**: `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:

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

Calling/Applying:

```python
>>> 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)**

Call expression:

- **square(2+2)**

Calling/Applying:

- **Square (x):**
  - **4**
  - **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:

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

Call expression:

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

Calling/Applying:

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

What happens?

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

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

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

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

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

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

What happens?
- A new frame is created!
- Argument
- Return value
Life Cycle of a User-Defined Function

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

**Call expression:**
- *Operator:* `square` (function: `func square(x)`)
- *Operand:* `2+2`
- *Argument:* `4`
- *Function (value of operator):* Called on arguments (values of operands)

**Calling/Applying:**
- *Argument:* 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:
- Name: `square(x):`
- Body (return statement): `return mul(x, x)`

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

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

Interactive Diagram
Multiple Environments in One Diagram!

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

Interactive Diagram
Multiple Environments in One Diagram!

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

Interactive Diagram
Multiple Environments in One Diagram!

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

Interactive Diagram
Multiple Environments in One Diagram!

Interactive Diagram

```
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))
```

**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!

```
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!

```
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!

```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

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

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

```
func square(x)
```

```
square(3)
```

```
func square(x)
```

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

```
81
```

```
9
```

```
3
```

Interactive Diagram
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

Interactive Diagram
Names Have No Meaning Without Environments

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.

Interactive Diagram

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.

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.

Interactive Diagram

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

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

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

**Compound statements:**

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

<separating header>:
  <statement>
  <statement>
  ...
  ...
```
A \textit{statement} is executed by the interpreter to perform an action

\textbf{Compound statements:}

\begin{itemize}
  \item \textbf{Statement}
  \item \textbf{Clause}
  \item \textbf{Suite}
\end{itemize}

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

**Compound statements:**

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

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>
...

...
Compound Statements

Compound statements:

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

Compound statements:

\[\text{<header>:} \quad \text{<statement>}
\]
\[\text{<statement>}
\]
\[\ldots\]

\[\text{<separating header>:} \quad \text{<statement>}
\]
\[\text{<statement>}
\]
\[\ldots\]

Execution Rule for a sequence of statements:

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

A suite is a sequence of statements

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

(Demo)
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:

1. Evaluate the header's expression.

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

(1) statement,
(1) clause,
(1) header,
(1) suite

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.

**Execution Rule for Conditional Statements:**

**Syntax Tips:**

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

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.

```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
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

George Boole

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

Boolean Contexts

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

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
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  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)

```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

George Boole

(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
```
While Statements

George Boole

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.
While Statements

George Boole

Execution Rule for While Statements:

1. Evaluate the header’s expression.

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

(Demo)

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

Global frame

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

(Demo)

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

Global frame

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

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

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>1</td>
</tr>
<tr>
<td>total</td>
<td>0</td>
</tr>
</tbody>
</table>

Execution Rule for While Statements:
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
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

George Boole

### Execution Rule for While Statements:

1. Evaluate the header’s expression.

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

```python
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.

(Demo)

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

George Boole

Execution Rule for While Statements:

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

(Demo)

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

Global frame

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

1. $i, \text{total} = 0, 0$
2. \textbf{while} $i < 3$: 
3. \hspace{1cm} $i = i + 1$
4. \hspace{1cm} $\text{total} = \text{total} + i$

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.

(Demo)

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

(Demo)

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

Global frame

- i: 3
- total: 3
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.

(i, total) = (0, 0)
while i < 3:
    i = i + 1
    total = total + i

(Global frame)

```
0 1 2 3
```

```
0 1 2 3
```
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.

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

(Demo)

Global frame
- i: 1, 2, 3
- total: 1, 2, 3, 6