Life Cycle of a User-Defined Function

Def statement:

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

What happens?

Call expression: `square(2+2)`

Calling/Applying:

```python
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?

Call expression: `square(2+2)`

Calling/Applying:

```python
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?

Call expression: `square(2+2)`

Calling/Applying:

```python
square( x ):
    return mul(x, x)
```
Life Cycle of a User-Defined Function

Def statement:

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

Calling/Applying:

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

What happens?
Life Cycle of a User-Defined Function

Def statement:

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

Calling/Applying:

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

What happens?
Life Cycle of a User-Defined Function

Def statement:

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

Call expression: `square(2+2)`

Calling/Applying:

```python
square( x ):
    return mul(x, x)
```
Life Cycle of a User-Defined Function

Def statement:  

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

What happens?  
Function created

Call expression:  square(2+2)

Calling/Applying:  

\[
square(x) = \text{mul}(x, x)
\]
Life Cycle of a User-Defined Function

Def statement:

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

Calling/Applying:

square(2+2)
Life Cycle of a User-Defined Function

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

Call expression: square(2+2)

What happens?
Function created
Name bound

Def statement

Formal parameter

Return expression

Body (return statement)

Calling/Applying:

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

Operand: 2+2
Argument: 4
Def statement:

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

Call expression:

operator: square
function: square
operand: 2+2
argument: 4

What happens?
Function created
Name bound

Calling/Applying:

```
square(x):
    return mul(x, x)
```
Life Cycle of a User-Defined Function

Def statement:

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

Calling/Applying:

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

What happens?

- Function created
- Name bound
- Op's evaluated
- Operand: 2+2
- Argument: 4
- Return expression: `mul(x, x)`
Life Cycle of a User-Defined Function

Def statement:

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

What happens?

Function created
Name bound

Op's evaluated
Function called with argument(s)

Call expression:

operator: square
function: square

Operand: 2+2
Argument: 4

Formal parameter

Body (return statement)

Calling/Applying:

```python
square(x):
    return mul(x, x)
```
Life Cycle of a User-Defined Function

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

**Formal parameter**

**Return expression**

**Body (return statement)**

**What happens?**
Function created
Name bound

**Call expression:**
square(2+2)
operator: square
function: square

**Operand:** 2+2
**Argument:** 4

**Op's evaluated**
Function called with argument(s)

**Calling/Applying:**
square(x):
return mul(x, x)

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

Def statement:

`>>> def square(x):`

```
    return mul(x, x)
```

Calling/Applying:

`square(2+2)`

What happens?

Function created

Name bound

Op's evaluated

Function called with argument(s)
Life Cycle of a User-Defined Function

Def statement:

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

Calling/Applying:

```python
>>> square(2+2)
```

What happens?
Function created
Name bound

Op's evaluated
Function called with argument(s)

Signature:
```
4 ➩ square(x):
    return mul(x, x) ➩ 16
```
Life Cycle of a User-Defined Function

Def statement:

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

Calling/Applying:

4 ▶️ square(x):
    return mul(x, x) ▶️ 16

What happens?

Function created
Name bound

Op's evaluated
Function called with argument(s)
Life Cycle of a User-Defined Function

Def statement:

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

Calling/Applying:

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

What happens?

Function created
Name bound
Op's evaluated
Function called with argument(s)
Life Cycle of a User-Defined Function

Def statement:

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

Calling/Applying:

Def statement

Formal parameter

Return expression

Body (return statement)

What happens?

Function created
Name bound

Op's evaluated
Function called with argument(s)

New frame!

operator: square
function: square

operand: 2+2
argument: 4

Argument

Signature

Return value
Life Cycle of a User-Defined Function

Def statement:

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

Calling/Applying:

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

What happens?

- Function created
- Name bound

Op's evaluated

- Operator: square
- Function: square
- Operand: 2+2
- Argument: 4

Function called with argument(s)

New frame!

- Argument: 4
- Signature: `square(x):`
- Return value: 16
- Return expression: `mul(x, x)`
- Body (return statement)
- Function created
- Name bound

Def statement:
Life Cycle of a User-Defined Function

Def statement:

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

What happens?
Function created
Name bound

Call expression:

Operator: square
Function: square

 operand: 2+2
argument: 4

Op's evaluated
Function called with argument(s)

Calling/Applying:

Signature
Argument: 4

Function created
Name bound

Formal parameter

Return expression

Body (return statement)

Return value

Body executed

New frame!

Argument

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4

4
Life Cycle of a User-Defined Function

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

- **Call expression:**
  - Operator: `square`
  - Function: `square`
  - Operand: `2+2`
  - Argument: `4`

- **Calling/Applying:**
  - Argument: `4`
  - Signature: `4`
  - Return value: `16`

**What happens?**
- Function created
- Name bound
- Op's evaluated
- Function called with argument(s)
- New frame!
- Params bound
- Body executed
Multiple Environments in One Diagram!

Example: http://goo.gl/668fU
Multiple Environments in One Diagram!

An environment is a sequence of frames.

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

Example: http://goo.gl/668fU

1 from operator import mul
2 def square(x):
3     return mul(x, x)
4     square(square(3))
Names Have No Meaning Without Environments

Every expression is evaluated in the context of an environment.

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

```
from operator import mul

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

Example: http://goo.gl/668fU
Formal Parameters

Example: http://goo.gl/OapJa
Formal Parameters

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

Example: http://goo.gl/OapJa
Formal Parameters

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

Example: [http://goo.gl/OapJa](http://goo.gl/OapJa)
Formal Parameters

def square(x):
    return mul(x, x)  # vs  def square(y):
    return mul(y, y)
Formal Parameters

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

vs

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

Example: [http://goo.gl/OapJa](http://goo.gl/OapJa)
Formal Parameters

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

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

Example: http://goo.gl/OapJa
Formal Parameters

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

**vs**

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

---

Example: [http://goo.gl/OapJa](http://goo.gl/OapJa)
Formal Parameters

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

vs
def square(y):
    return mul(y, y)
```

Formal parameters have local scope

Example: [http://goo.gl/OapJa](http://goo.gl/OapJa)
Python Feature Demonstration

<Demo>

Operators

Multiple Return Values

Docstrings

Doctests

Default Arguments

Statements

</Demo>
A statement is executed by the interpreter to perform an action.
A statement is executed by the interpret to perform an action

Compound statements:

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

Compound statements:

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

**Compound statements:**

```plaintext
<header>:
  <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>
  ...
  ...
```
Statements

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.
Statements

A statement is executed by the interpret 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:

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

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

Suite
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
• Unless directed otherwise, execute the rest
Local Assignment

def percent_difference(x, y):
    difference = abs(x-y)
    return 100 * difference / x

percent_difference(40, 50)
def percent_difference(x, y):
    difference = abs(x - y)
    return 100 * difference / x

percent_difference(40, 50)
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

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 rest.
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

Two boolean contexts

George Boole
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

Two boolean contexts
Boolean Contexts

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

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

False values in Python: False, 0, '', None
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

Two 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

Two boolean contexts

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

True values in Python: Anything else (True)
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

Two boolean contexts

George Boole

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

True values in Python:    Anything else (True)

Read Section 1.5.4!
Iteration

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

```
0 1 2 3 6
```
**Iteration**

\[\begin{align*}
i, \text{total} &= 0, 0 \\
\text{while } i < 3: & \\
i &= i + 1 \\
\text{total} &= \text{total} + i
\end{align*}\]

**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.
Iteration

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.
Iteration

```
▶ 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.
**Iteration**

```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.
Iteration

```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.
Iteration

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.
Iteration

----------

\[
i, \text{total} = 0, 0
\]

\[
\text{while } i < 3:\n\]
\[
i = i + 1
\]
\[
\text{total} = \text{total} + i
\]

 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.
### Iteration

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

```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.
Iteration

```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.
Iteration

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

<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.
Iteration

```
\[ i, \text{ total} = 0, 0 \]
\[ \textbf{while } i < 3: \]
\[ \quad i = i + 1 \]
\[ \quad \text{total} = \text{total} + i \]
```

```
<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.
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.

\[ i, \text{ total} = 0, 0 \]
\[ \text{while } i < 3: \]
\[ i = i + 1 \]
\[ \text{total} = \text{total} + i \]
**Iteration**

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

<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.
Iteration

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.
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, 
exceute the (whole) suite, 
then return to step 1.
Iteration

```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.
**Iteration**

```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.
The Fibonacci Sequence
The Fibonacci Sequence

0, 1, 1, 2, 3, 5, 8, 13, ...
def fib(n):
    """Compute the nth Fibonacci number, for n >= 2."""
    pred, curr = 0, 1  # First two Fibonacci numbers
    k = 2  # Tracks which Fib number is curr
    while k < n:
        pred, curr = curr, pred + curr
        k = k + 1
    return curr
The Fibonacci Sequence

```python
def fib(n):
    """Compute the nth Fibonacci number, for n >= 2."""
    pred, curr = 0, 1  # First two Fibonacci numbers
    k = 2              # Tracks which Fib number is curr
    while k < n:
        pred, curr = curr, pred + curr
        k = k + 1
    return curr
```

0, 1, 1, 2, 3, 5, 8, 13, ...
The Fibonacci Sequence

def fib(n):
    """Compute the nth Fibonacci number, for n >= 2."""
    pred, curr = 0, 1  # First two Fibonacci numbers
    k = 2  # Tracks which Fib number is curr
    while k < n:
        pred, curr = curr, pred + curr
        k = k + 1
    return curr
The Fibonacci Sequence

```
def fib(n):
    """Compute the nth Fibonacci number, for n >= 2."""
    pred, curr = 0, 1  # First two Fibonacci numbers
    k = 2  # Tracks which Fib number is curr
    while k < n:
        pred, curr = curr, pred + curr
        k = k + 1
    return curr
```
The Fibonacci Sequence

def fib(n):
    """Compute the nth Fibonacci number, for n >= 2."""
    pred, curr = 0, 1  # First two Fibonacci numbers
    k = 2              # Tracks which Fib number is curr
    while k < n:
        pred, curr = curr, pred + curr
        k = k + 1
    return curr

0, 1, 1, 2, 3, 5, 8, 13, ...
The Fibonacci Sequence

def fib(n):
    """Compute the nth Fibonacci number, for n >= 2."""
    pred, curr = 0, 1  # First two Fibonacci numbers
    k = 2             # Tracks which Fib number is curr
    while k < n:
        pred, curr = curr, pred + curr
        k = k + 1
    return curr
The Fibonacci Sequence

... pred: curr:

\[0, 1, 1, 2, 3, 5, 8, 13, \ldots\]

def fib(n):
    """Compute the nth Fibonacci number, for n >= 2."""
    pred, curr = 0, 1  # First two Fibonacci numbers
    k = 2  # Tracks which Fib number is curr
    while k < n:
        pred, curr = curr, pred + curr
        k = k + 1
    return curr
def fib(n):
    """Compute the nth Fibonacci number, for n >= 2."""
    pred, curr = 0, 1  # First two Fibonacci numbers
    k = 2              # Tracks which Fib number is curr
    while k < n:
        pred, curr = curr, pred + curr
        k = k + 1
    return curr
Project 1: Hog

(Demo)