None Indicates that Nothing is Returned

The special value `None` represents nothing in Python.

A function that does not explicitly return a value will return `None`.

Careful: `None` is not displayed by the interpreter as the value of an expression.

```python
>>> def does_not_return_square(x):
    ...
    x * x
    ...
    # No return

>>> does_not_return_square(4)  # None value is not displayed
>>> 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`.
Pure Functions & Non-Pure Functions

**Pure Functions**
- just return values
  - `abs(-2)` returns `2`

**Non-Pure Functions**
- have side effects
  - `print(-2)` displays the output “-2”

Nested Expressions with Print

```
None, None >>> print(print(1), print(2))
Does not get displayed

display "None None"

func print(...) None

func print(...) 2 print(...): None

func print(...) 2

1 print(...): None

display "1"

2 print(...): None

display "2"
```

Life Cycle of a User-Defined Function

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

Formal parameter

Return expression

Body (return statement)

operand: 2+2 argument: 4

operator: square function: func square(x)

Call expression: square(2+2) operand: 2+2 argument: 4

Calling/Applying: 4 square(x): 16

Argument

Signature

Return value

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
Multiple Environments in One Diagram!

An environment is a sequence of frames.
- The global frame alone
- A local, then the global frame

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.

Conditional 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.

Miscellaneous Python Features

Division
Multiple Return Values
Source Files
Doctests
Default Arguments
(Demo)
Compound Statements

Compound statements:

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

A suite is a sequence of statements.
To "execute" a suite means to execute its sequence of statements, in order.

Execution Rule for a sequence of statements:
- Execute the first statement
- Unless directed otherwise, execute the rest

Conditional Statements

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

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.
- 3. Zero or one "else" clause, always at the end.

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

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

Read Section 1.5.4!

(Demo)
Iteration

While Statements

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>i, total = 0, 0</td>
<td>Initial values</td>
</tr>
<tr>
<td>2</td>
<td>while i &lt; 3</td>
<td>Loop condition</td>
</tr>
<tr>
<td>3</td>
<td>i = i + 1</td>
<td>Increment i</td>
</tr>
<tr>
<td>4</td>
<td>total = total + i</td>
<td>Accumulate total</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.

Example: Prime Factorization

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

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

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

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

(Demo)