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
...     4
...     # No return

>>> does_not_return_square(4)
The name sixteen is now bound to the value 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'
```

Pure Functions & Non-Pure Functions

**Pure Functions**

- `abs` just return values
- `pow` Arguments
- `print` Returns None

**Non-Pure Functions**

- `print` have side effects
- `abs`, `pow` Arguments

```
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

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

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

1 >>> print(print(1), print(2))
display "1"

2 >>> print(print(1), print(2))
display "2"
```
Life Cycle of a User-Defined Function

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

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

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

Statements

A statement is executed by the interpreter to perform an action

Conditional Statements

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**

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**

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**:
- Always starts with "if" clause.
- Zero or more "elif" clauses.
- Zero or one "else" clause, always at the end.

**Boolean Contexts**

- **False values in Python**: False, 0, '', None
- **True values in Python**: Anything else (True)

Read Section 1.5.4!
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.

Iteration Example

The Fibonacci Sequence

```
def fib(n):
    """Compute the nth Fibonacci number, for N >= 1."""
    pred, curr = 0, 1  # 0th and 1st Fibonacci numbers
    k = 1  # curr is the kth Fibonacci number
    while k < n:
        pred, curr = curr, pred + curr
        k = k + 1
    return curr
```

The next Fibonacci number is the sum of the current one and its predecessor.
Return Statements

A return statement completes the evaluation of a call expression and provides its value:
- \( f(x) \) for user-defined function \( f \): switch to a new environment; execute \( f \)'s body
- return statement within \( f \): switch back to the previous environment; \( f(x) \) now has a value

Only one return statement is ever executed while executing the body of a function

```python
def end(n, d):
    # Print the final digits of N in reverse order until D is found.
    while n > 0:
        last, n = n % 10, n // 10
        print(last)
        if d == last:
            return None
```

Designing Functions
Describing Functions

A function's **domain** is the set of all inputs it might possibly take as arguments.

A function's **range** is the set of output values it might possibly return.

A pure function's **behavior** is the relationship it creates between input and output.

```python
def square(x):
    """Return X * X."""
    x is a number
    square returns a non-negative real number
    square returns the square of x
```

A Guide to Designing Function

Give each function exactly one job, but make it apply to many related situations

```python
>>> round(1.23)  # round to nearest whole number
1
>>> round(1.23, 1)  # round to one decimal place
1.2
>>> round(1.23, 0)  # round to nearest whole number
1
>>> round(1.23, 5)  # round to five decimal places
1.23
```

Don’t repeat yourself (DRY): Implement a process just once, but execute it many times

(Demo)

Generalizing Patterns with Arguments

Regular geometric shapes relate length and area.

Finding common structure allows for shared implementation

(Demo)
If Statements and Call Expressions

Let's try to write a function that does the same thing as an if statement.

Execution Rule for Conditional Statements:
Each clause is considered in order.
1. Evaluate the header's expression (if present).
2. If it is a true value (or an else header), execute the suite & skip the remaining clauses.

Evaluation Rule for Call Expressions:
1. Evaluate the operator and then the operand subexpressions
2. Apply the function that is the value of the operator to the arguments that are the values of the operands

Logical Operators

To evaluate the expression \(<left>\text{and}\ <right>\):
1. Evaluate the subexpression \(<left>\).
2. If the result is a false value \(v\), then the expression evaluates to \(v\).
3. Otherwise, the expression evaluates to the value of the subexpression \(<right>\).

To evaluate the expression \(<left>\text{or}\ <right>\):
1. Evaluate the subexpression \(<left>\).
2. If the result is a true value \(v\), then the expression evaluates to \(v\).
3. Otherwise, the expression evaluates to the value of the subexpression \(<right>\).
Conditional Expressions

A conditional expression has the form

<consequent> if <predicate> else <alternative>

Evaluation rule:
1. Evaluate the <predicate> expression.
2. If it's a true value, the value of the whole expression is the value of the <consequent>.
3. Otherwise, the value of the whole expression is the value of the <alternative>.

```python
>>> x = 0
>>> abs(1/x if x != 0 else 0)
0
```