```
def fib(n):
    a, b = 0.0, 1.0
    for i in range(n):
        a, b = a + b, a
    return a
```

As mentioned in the introduction, this Python function is already a valid Cython function, and it has identical behavior in both Python and Cython. We will see shortly how we can add Cython-specific syntax to `fib` to improve its performance.

The C transliteration of `fib` follows the Python version closely:

```
double cfib(int n) {
    int i;
    double a=0.0, b=1.0, tmp;
    for (i=0; i<n; ++i) {
        tmp = a; a = a + b; b = tmp;
    }
    return a;
}
```

We use `doubles` in the C version and `floats` in the Python version to make the comparison direct and remove any issues related to integer overflow for C integral data types.

Imagine blending the types from the C version with the code from the Python version. The result is a statically typed Cython version:

```
def fib(int n):
    cdef int i
    cdef double a=0.0, b=1.0
    for i in range(n):
        a, b = a + b, a
    return a
```

As mentioned previously, Cython understands Python code, so our unmodified Python `fib` function is also valid Cython code. To convert the dynamically typed Python version to the statically typed Cython version, we use the `cdef` Cython statement to declare the statically typed C variables `i`, `a`, and `b`. Even for readers who haven't seen Cython code before, it should be straightforward to understand what is going on.

What about performance? Table 1-1 has the results.

*Table 1-1. Fibonacci timings for different implementations*

| Version | fib(0) [ns] | Speedup | fib(90) [ns] | Speedup | Loop body [ns] | Speedup |
|---|---|---|---|---|---|---|
| Pure Python | 590 | 1 | 12,852 | 1 | 12,262 | 1 |
| Pure C | 2 | 295 | 164 | 78 | 162 | 76 |
| C extension | 220 | 3 | 386 | 33 | 166 | 74 |
| Cython | 90 | 7 | 258 | 50 | 168 | 73 |