The process of converting Python code to machine code involves several layers of abstraction and compilation. Here's a step-by-step explanation, along with an example at each layer:

1. Python Code

2. Python Compilation to Bytecode

3. Intermediate Representation (C Code)

4. Compilation to Machine Code

5. Machine Code

```
graph TD

subgraph cluster_python_code
  style PythonCodeStyle fill:#e6f7ff,stroke:#4d94ff,stroke-width:2px;
  a[Python Code]
end

subgraph cluster_bytecode
  style BytecodeStyle fill:#e6f7ff,stroke:#4d94ff,stroke-width:2px;
  b[Bytecode]
end

subgraph cluster_c_code
  style CCodeStyle fill:#e6f7ff,stroke:#4d94ff,stroke-width:2px;
  c[C Code]
end

subgraph cluster_assembly_code
  style AssemblyCodeStyle fill:#e6f7ff,stroke:#4d94ff,stroke-width:2px;
  d[Assembly Code]
end

subgraph cluster_machine_code
  style MachineCodeStyle fill:#e6f7ff,stroke:#4d94ff,stroke-width:2px;
  e[Machine Code]
end

a -->|Run| b
b -->|Cython| c
c -->|Compilation| d
d -->|Assembly| e
```

## 1. Python Code

```python
# Example Python Code
def add_numbers(a, b):
    return a + b

result = add_numbers(5, 7)
print(result)
```

## 2. Python Compilation to Bytecode

When you run a Python script, the Python interpreter compiles the code into bytecode. Bytecode is an intermediate representation of the code that is platform-independent.

```
# Example Bytecode (not the actual bytecode, just for illustration)
LOAD_CONST 5       # Load constant 5 onto the stack
LOAD_CONST 7       # Load constant 7 onto the stack
BINARY_ADD         # Add the top two stack items
STORE_NAME result # Store the result in the 'result' variable
LOAD_NAME print    # Load the 'print' function
LOAD_NAME result   # Load the 'result' variable
CALL_FUNCTION 1    # Call the 'print' function with 1 argument
```

## 3. Intermediate Representation (C Code)

Tools like Cython can convert Python bytecode into C code. This involves translating the Python code into equivalent C code.

```c
// Example C Code (generated by Cython, simplified for illustration)
#include <stdio.h>

int add_numbers(int a, int b) {
    return a + b;
}

int main() {
    int result = add_numbers(5, 7);
    printf("%d\n", result);
    return 0;
}
```

## 4. Compilation to Machine Code

The C code is then compiled into machine code, which is specific to the target architecture (e.g., x86, ARM).

```
; Example Assembly Code (simplified for illustration, x86 architecture)
section .data
section .text
    global add_numbers
    add_numbers:
        mov eax, [esp + 4]   ; Load the first argument into eax
        add eax, [esp + 8]   ; Add the second argument
        ret

    global _start
    _start:
        push 7               ; Push second argument onto the stack
        push 5               ; Push first argument onto the stack
        call add_numbers     ; Call the add_numbers function
        add esp, 8           ; Adjust the stack pointer
        mov ebx, eax         ; Move the result to ebx
        mov eax, 4           ; System call for sys_write
        mov ecx, ebx         ; Address of the result
        int 0x80             ; Make system call
        mov eax, 1           ; System call for sys_exit
        xor ebx, ebx         ; Exit code 0
        int 0x80             ; Make system call
```

## 5. Machine Code

The assembly code is then assembled into machine code, which consists of binary instructions that the computer's CPU can directly execute.

```
; Example Machine Code (hexadecimal representation)
55 8B EC 83 EC 08 68 07 00 00 00 68 05 00 00 00 E8 00 00 00 00 83 C4 08 89 C3 B8 04 00 00 00 B9 00 00 00 00 CD 80 B8 01 00 00 00 31 DB CD 80
```

This machine code can be executed directly by the computer's CPU, performing the addition and printing the result to the console.

It's important to note that the process may vary, and there are optimizations and additional steps involved in real-world scenarios. This example provides a simplified overview of the general process.