C++ & Python Program Design
-- Basics
Interact with Your Program

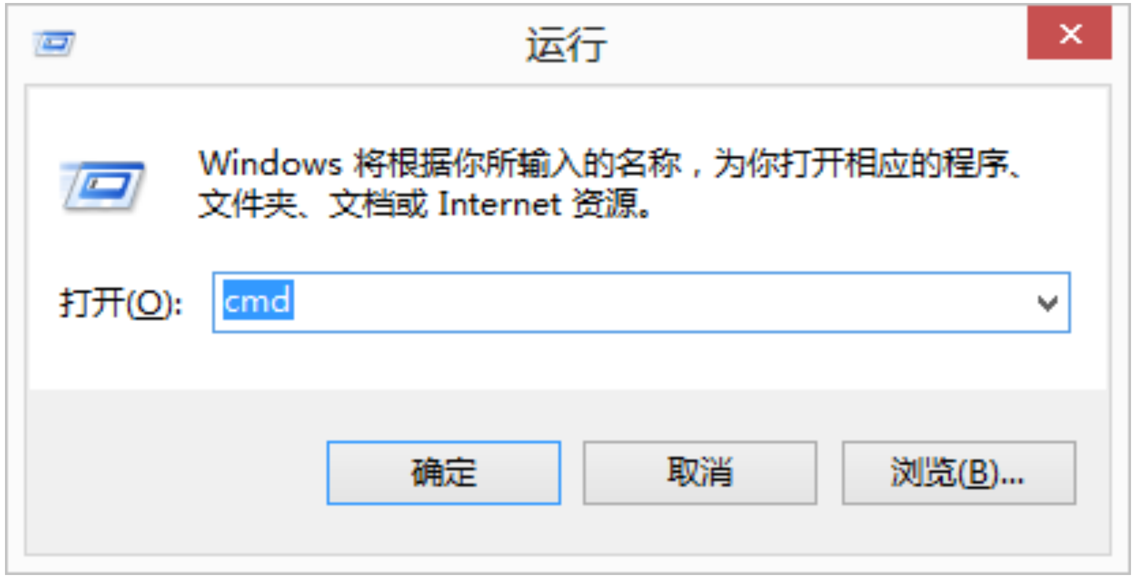
Junjie Cao @ DLUT

Summer 2022

https://github.com/jjcao-school/c

Command line interface (CLI) 命令行接口

- provides a way for a user to interact with a program running in a text-based shell interpreter.
  - Command line arguments
  - cout & cin
  - print & input

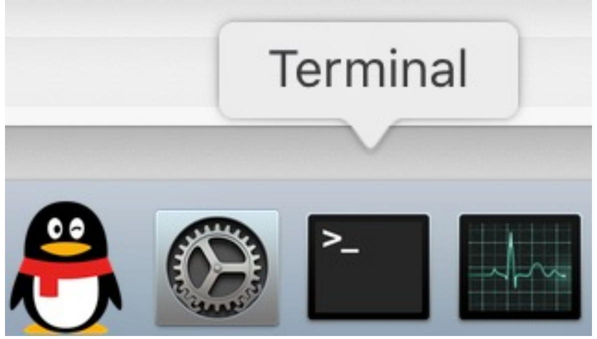


运行

Windows 将根据你所输入的名称，为你打开相应的程序、文件夹、文档或 Internet 资源。

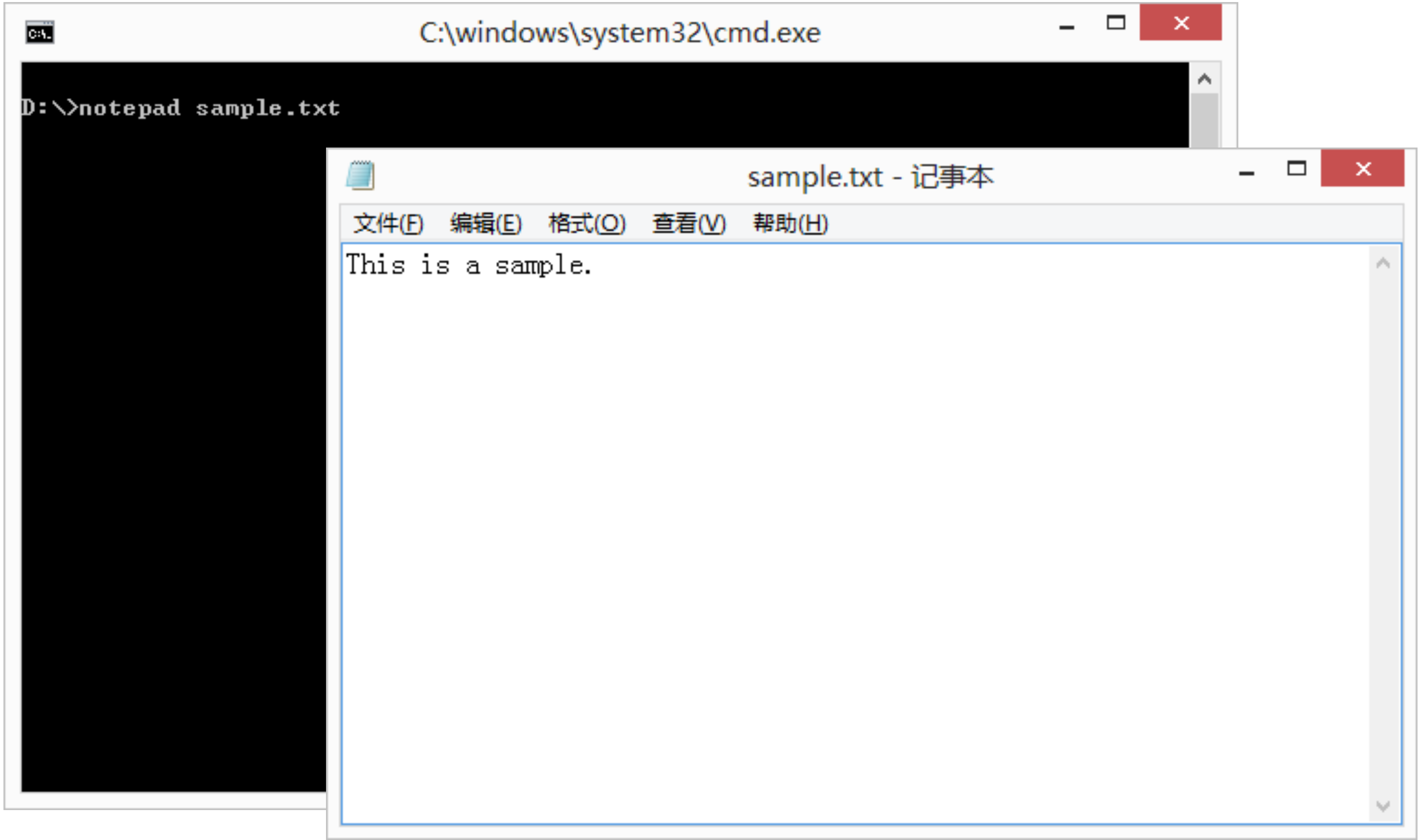
打开(O): cmd

Windows + R 键：



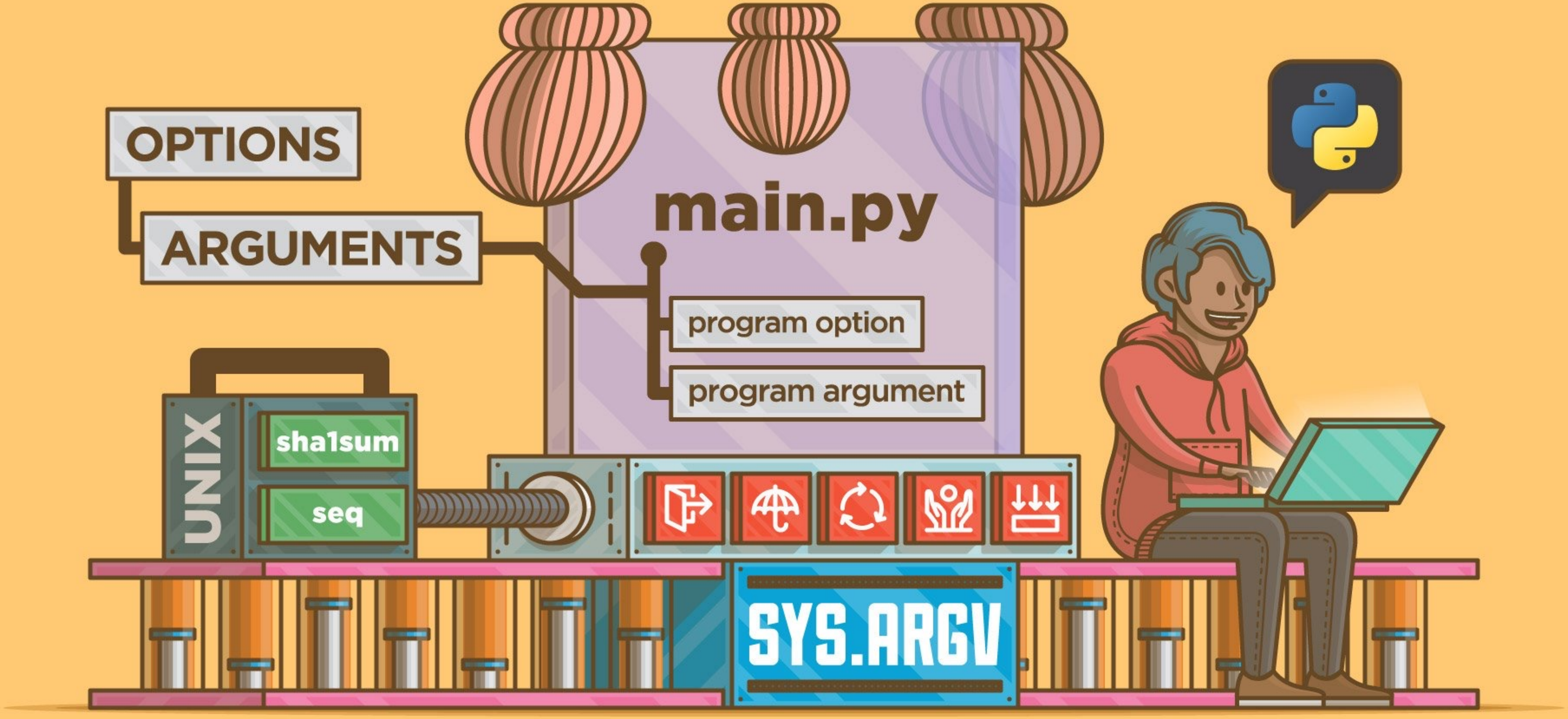
Terminal on Mac：

命令行方式运行程序



```
D:\>notepad sample.txt
```

```
This is a sample.
```



CLI &命令行参数

Real Python

命令行方式运行程序

```
notepad    sample.txt
```

notepad　程序如何得知，用户在以命令行方式运行它的时候，后面跟着什么参数？

命令行参数

将用户在CMD窗口输入可执行文件名的方式 启动程序时，跟在可执行文件名后面的那些 字符串，称为“命令行参数”。命令行参数

可以有多个，以空格分隔。比如，在CMD窗口敲：

```
copy   file1.txt file2.txt
```

"copy", "file1.txt", "file2.txt" 就是命令行参数

如何在程序中获得命令行参数呢？

命令行参数

```
int  main(int   argc,  char  * argv[])
{
        ......
}
```

- argc：代表启动程序时，命令行参数的个数。

- C/C++语言规定，可执行程序程序本身的文件名，也算一个命令行参数，
- 因此，argc的值 至少是1。

命令行参数

```
int  main(int   argc,   char  * argv[])
{
        ……
}
```

argc: 代表启动程序时，命令行参数的个数。C/C++语言规定，可 执行程序程序本身的文件名，也算一个命令行参数，因此，argc的值 至少是1。

argv: 指针数组，其中的每个元素都是一个char* 类型的指针，该指针指向一个字符串，这个字符串里就存放着命令行参数。

例如，argv[0]指向的字符串就是第一个命令行参数，即可执行程序的文件名，argv[1]指向第二个命令行参数，argv[2]指向第三个命令 行参数……。

```
#include   <stdio.h>
int  main(int   argc,  char  *  argv[])
{
      for(int  i  = 0;i  < argc;  i  ++ )
              printf(  "%s\n",argv[i]);  return     0;
}
```

将上面的程序编译成sample.exe，然后在控制台 窗口敲:

```
sample   para1   para2   s.txt   5 “hello   world”
```

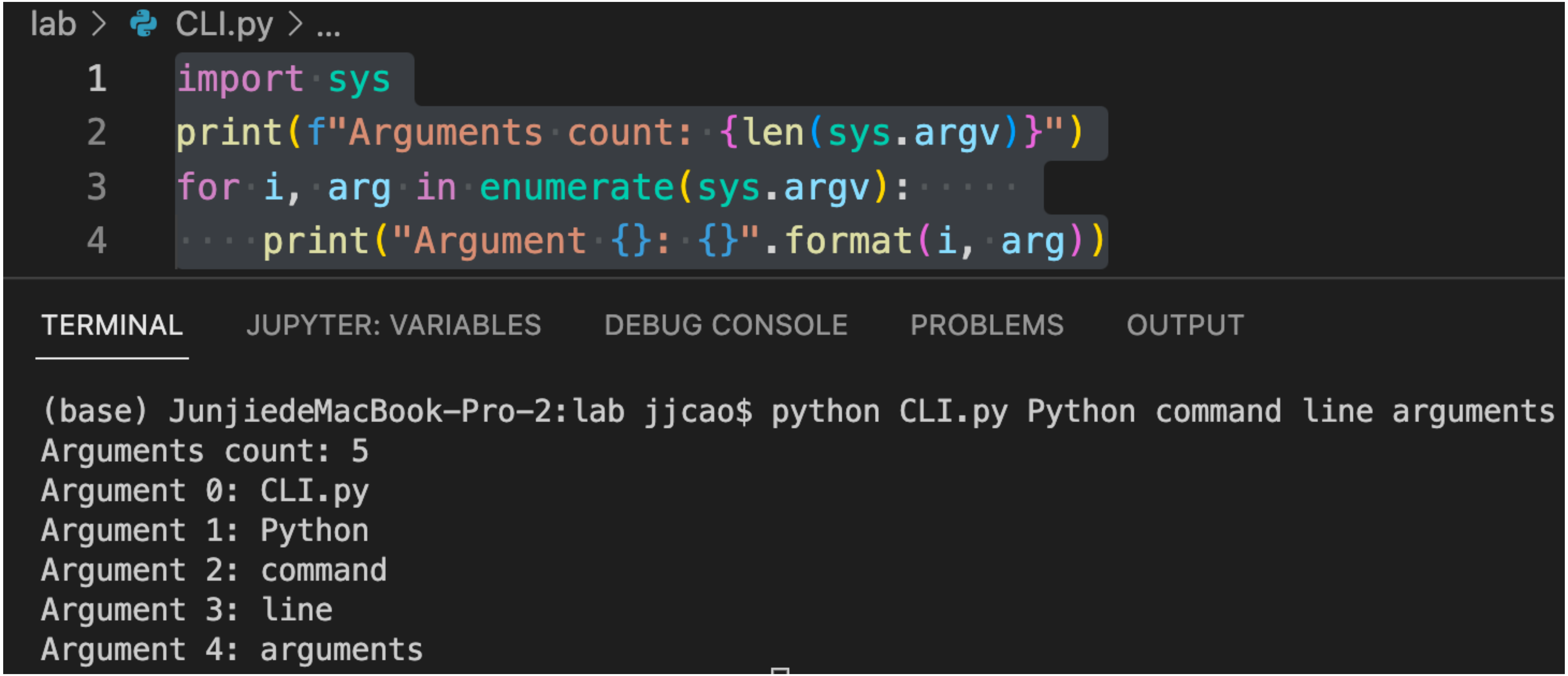
```
#include  <stdio.h>
int  main(int  argc,  char  *  argv[])
{
    for(int  i  = 0;i  < argc;  i  ++ )
          printf(  "%s\n",argv[i]);
    return  0;
}
```

将上面的程序编译成sample.exe，然后在控制台 窗口敲:

```
sample  para1  para2  s.txt  5 “hello  world”
```

输出结果就是：

```
sample
para1
para2
s.txt
5
hello  world
```



```python
import sys
print(f"Arguments count: {len(sys.argv)}")
for i, arg in enumerate(sys.argv):
    print("Argument {}: {}".format(i, arg))
```

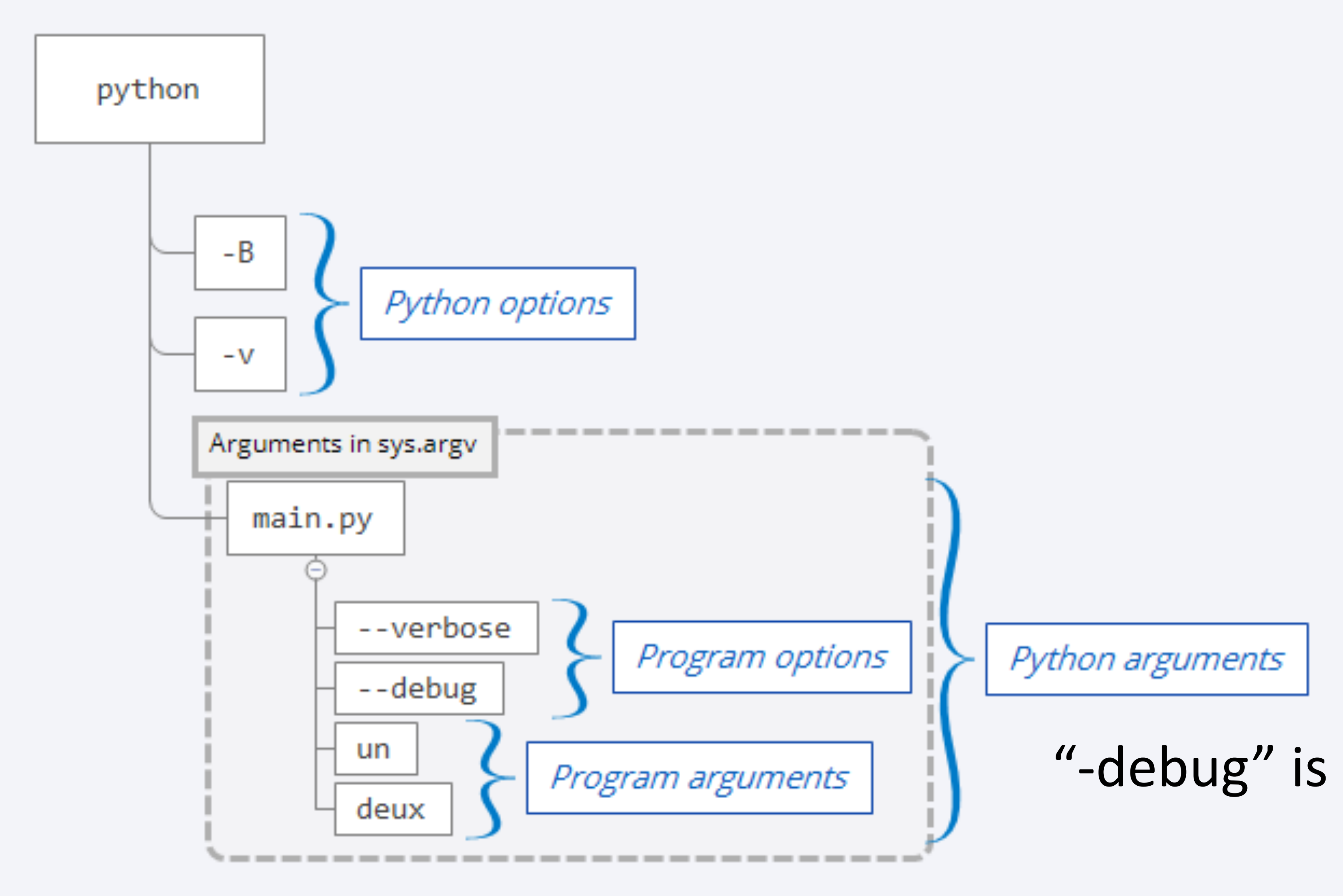
```
(base) JunjiedeMacBook-Pro-2:lab jjcao$ python CLI.py Python command line arguments
Arguments count: 5
Argument 0: CLI.py
Argument 1: Python
Argument 2: command
Argument 3: line
Argument 4: arguments
```

```python
import sys
print(f"Name of the script : {sys.argv[0]=}")
print(f"Arguments of the script : {sys.argv[1:]=}")
```

```
>>> python CLI.py Python command line arguments
Name of the script      :  CLI.py
Arguments of the script :  ['Python', 'command', 'line', 'arguments']
```

Options & Arguments

- python -B -v main.py --verbose --debug un deux



“-debug” is OK too.

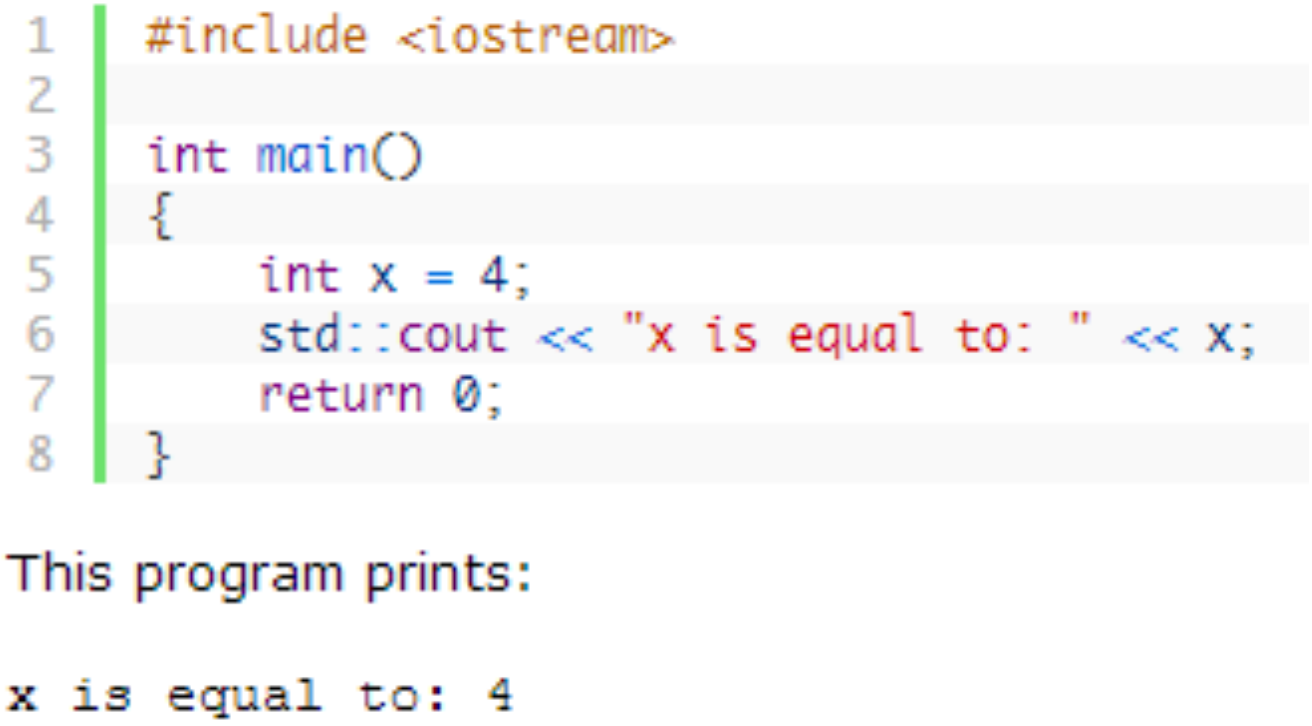
```
opts = [opt for opt in sys.argv[1:] if opt.startswith("-")]
args = [arg for arg in sys.argv[1:] if not arg.startswith("-")]

if "-c" or "--c" in opts:
print(" ".join(arg.capitalize() for arg in args))
elif "-u" in opts:
print(" ".join(arg.upper() for arg in args))
elif "-l" in opts:
print(" ".join(arg.lower() for arg in args))
else:
raise SystemExit(f"Usage: {sys.argv[0]} (-c | -u | -l) <arguments>...")
```

```
>>> python CLI.py --c un deux trois
```

cout & cin

std::cout

- the std::cout object (in the iostream library) can be used to output text to the console.
- To print more than one thing on the same line, the output operator (<<) can be used multiple times. For example:



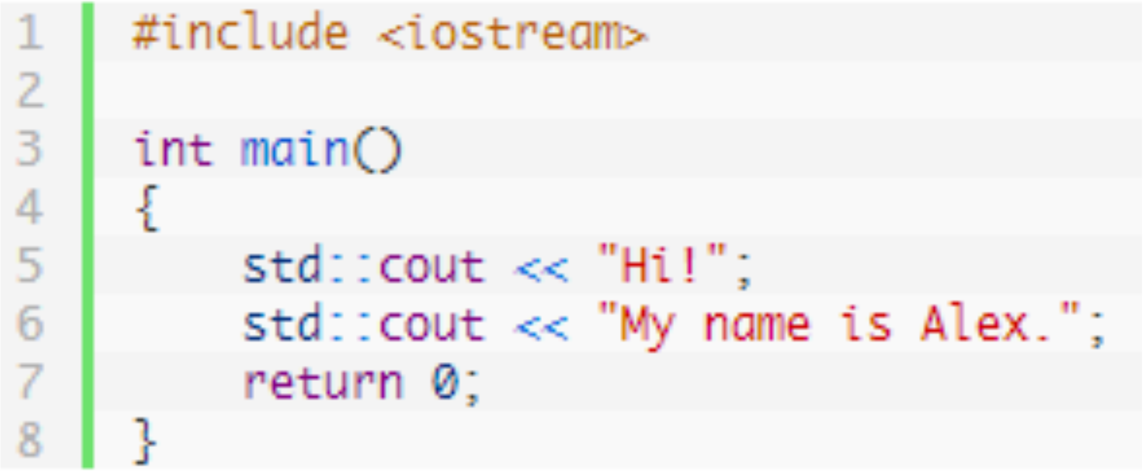
```cpp
#include <iostream>

int main()
{
    int x = 4;
    std::cout << "x is equal to: " << x;
    return 0;
}
```

This program prints:

```
x is equal to: 4
```

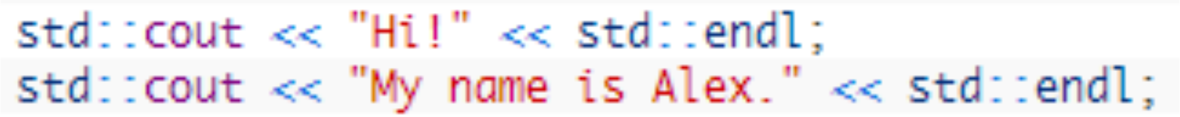
std::cout

- What would you expect this program to print?



```
#include <iostream>

int main()
{
    std::cout << "Hi!";
    std::cout << "My name is Alex.";
    return 0;
}
```

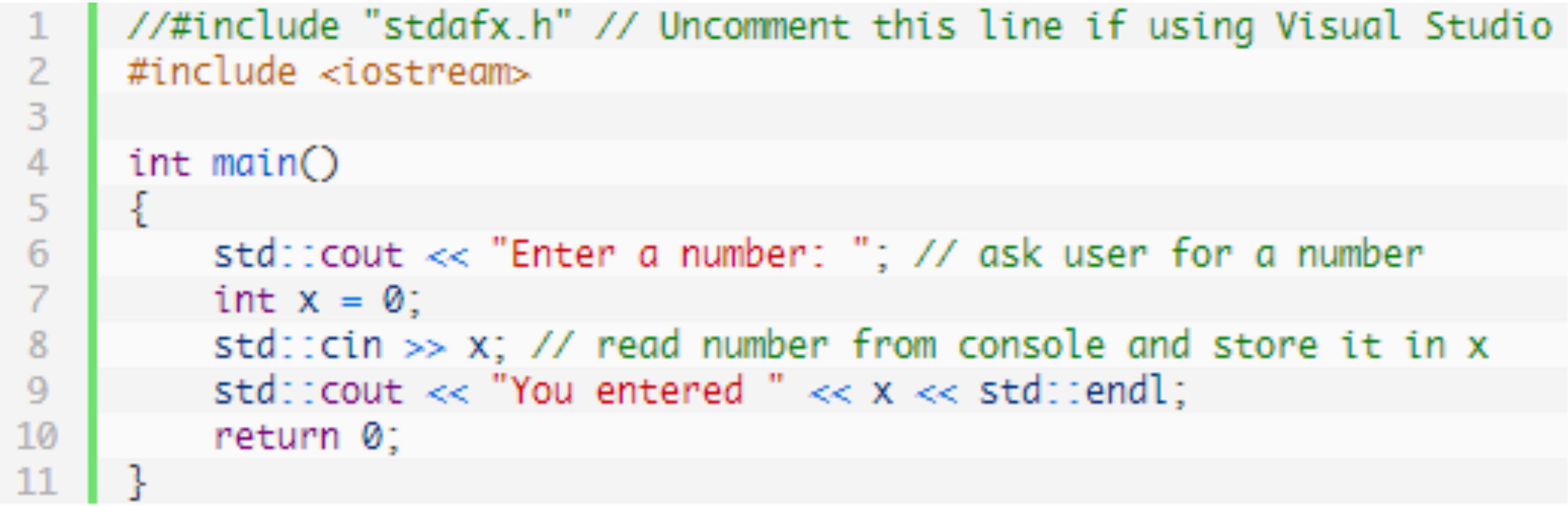
- Hi!My name is Alex.



```
std::cout << "Hi!" << std::endl;
std::cout << "My name is Alex." << std::endl;
```

std::cin

- std::cin reads input from the user at the console using the input operator (>>).
- we can store it in a variable.



```cpp
//#include "stdafx.h" // Uncomment this line if using Visual Studio
#include <iostream>

int main()
{
    std::cout << "Enter a number: "; // ask user for a number
    int x = 0;
    std::cin >> x; // read number from console and store it in x
    std::cout << "You entered " << x << std::endl;
    return 0;
}
```

```
Enter a number: 4

You entered 4
```

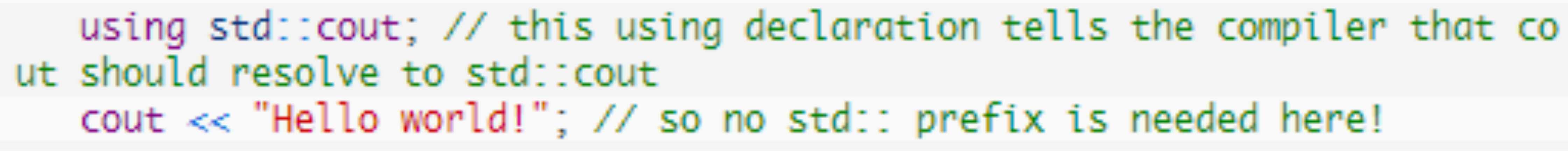
If your screen closes immediately after entering a number, add a statement:
std::cin.get(); or std::cin >>x;

After pressing “Enter” key, the console will be closed.

The std namespace

- Everything in the standard library is defined inside a special area (called a **namespace**) that is named *std* (short for standard).
  - It turns out that std::cout's name isn't really "std::cout". It's actually just "cout", and **"std" is the name of the namespace it lives inside**.
  - We'll talk more about namespaces in future and also teach you how to create your own.

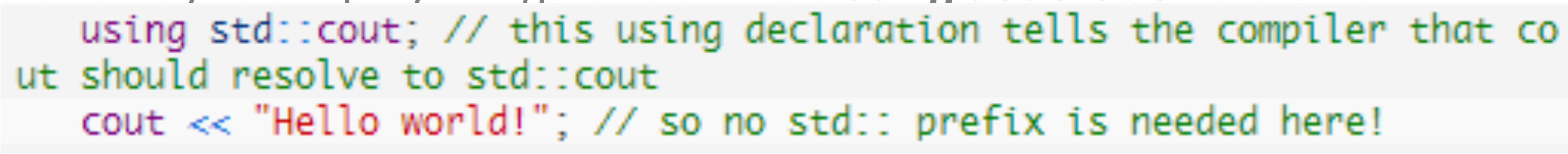
- whenever we use something (like cout) that is part of the standard library, we need to **tell the compiler that it is inside** the std namespace.
  - **Explicit namespace qualifier std::,** std::cout << "Hello world!";
  - One way to simplify things is to utilize a **using declaration** statement.



```
   using std::cout; // this using declaration tells the compiler that co
ut should resolve to std::cout
   cout << "Hello world!"; // so no std:: prefix is needed here!
```

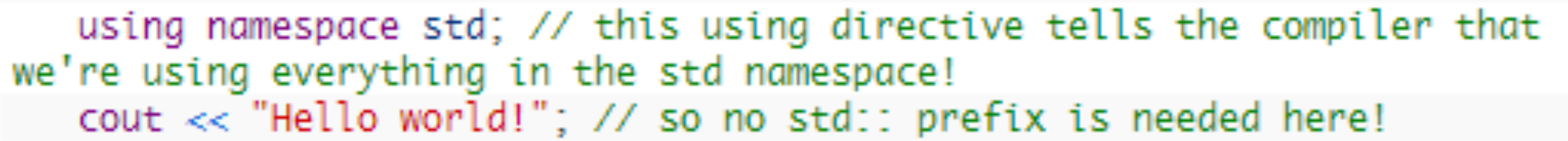
namespace

- whenever we use something (like cout) that is part of the standard library, we need to **tell the compiler that it is inside** the std namespace.
    - **Explicit namespace qualifier std::,** std::cout << "Hello world!";
    - One way to simplify things is to utilize a **using declaration** statement.



```
   using std::cout; // this using declaration tells the compiler that co
ut should resolve to std::cout
   cout << "Hello world!"; // so no std:: prefix is needed here!
```

- The **using directive**: tells the compiler that we want to use *everything* in the std namespace, so if the compiler finds a name it doesn't recognize, it will check the std

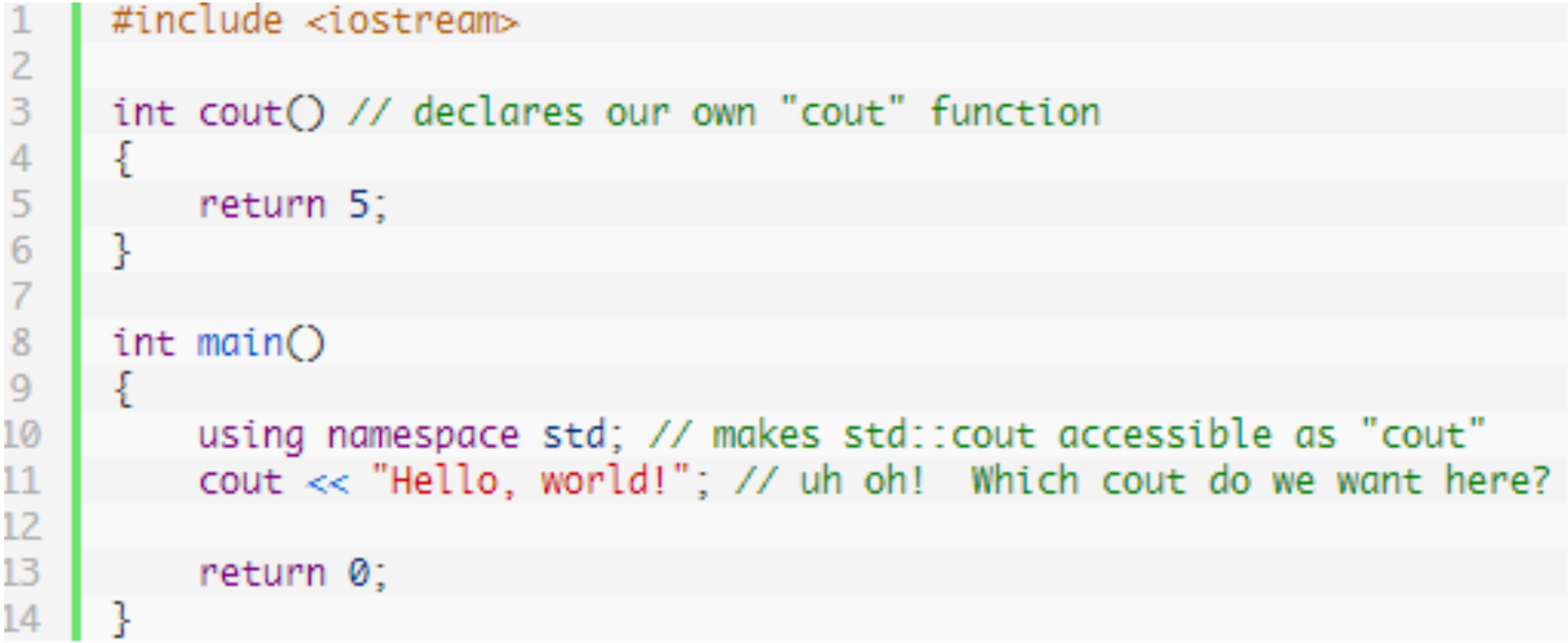


```
   using namespace std; // this using directive tells the compiler that
we're using everything in the std namespace!
   cout << "Hello world!"; // so no std:: prefix is needed here!
```

- a naming conflict => compile error

Using declarations and directives inside or outside of a function

- If a using declaration or directive is **used within a function**, the names in the namespace are only directly accessible within that function.
  - That limits the chance for naming collisions to occur just within that function.
- if a using declaration or directive is used **outside of a function**, the names in the namespace are directly accessible anywhere in the entire file!
  - That can greatly increase the chance for collisions.



```
#include <iostream>

int cout() // declares our own "cout" function
{
    return 5;
}

int main()
{
    using namespace std; // makes std::cout accessible as "cout"
    cout << "Hello, world!"; // uh oh!  Which cout do we want here?

    return 0;
}
```