CS100 Python
Introduction to Programming

Lecture 21. Introduction to Python

Fu Song

School of Information Science and Technology

ShanghaiTech University

About me

- Instructor: Fu Song

  Office: Room 1A-504.C, SIST Building

  Email: songfu@shanghaitech.edu.cn

Mission

develop theory and tools to
aid the construction of
provably dependable and secure systems

Course Materials

1. Goal: Programming in Python
2. Reference: https://www.python.org
   - The Python Tutorial
   - The Python Language Reference
   - The Python Standard Library
   - The Python HOWTOs
3. Lecture notes: Slides are available on Piazza

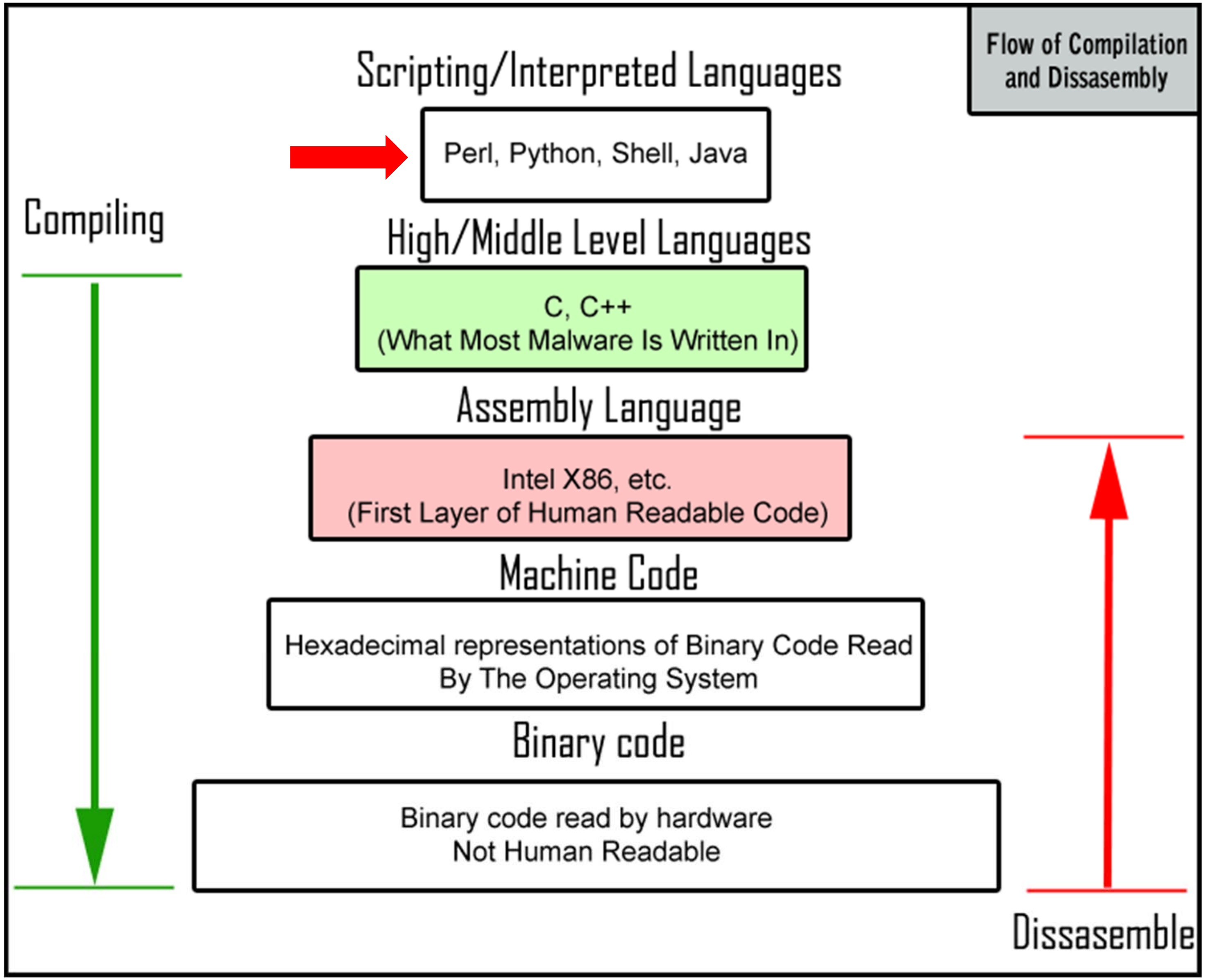
Plan for Learning Python

- Week 12
  - Introduction to Python: IO, diff C/C++ and Python
  - Basic Concepts: object, name, expression, control flow
- Week 13
  - Function and Scope
  - OOP in Python: Class
- Week 14
  - Sequence, Set and Mapping Types
  - Inheritance
- Week 15
  - Garbage Collection
  - Data Analytics and Visualization

Learning Objectives

- Understand when use
  - C/C++
  - Python
- Simple I/O in Python
- Understand the difference between
  - C/C++
  - Python

Programming Languages



Flow of Compilation and Dissasembly

Scripting/Interpreted Languages

Perl, Python, Shell, Java

High/Middle Level Languages

C, C++
(What Most Malware Is Written In)

Assembly Language

Intel X86, etc.
(First Layer of Human Readable Code)

Machine Code

Hexadecimal representations of Binary Code Read By The Operating System

Binary code

Binary code read by hardware
Not Human Readable

What makes a language successful?

- Expressive Power: easily solve complex problem
- Ease of Use for Novices: easy to learn
- Ease of Implementation: portable for platforms
- Open Source: C/C++, Java, Python, Rust, etc.
- Excellent Compilers: efficiency & effectiveness
- Supporter: supported by large and powerful organizations

  e.g. C# by Microsoft, Java by Oracle, Python by community

Some languages live on because of a large amount of legacy code

How to pick a language?

- Meet your application requirements
  - Must it be efficient?
  - Can it afford the runtime (e.g., garbage collector)?
  - Easy to understand
  - Easy to write (time, size of programs)
  - Easy to debug, maintain, and prevent errors

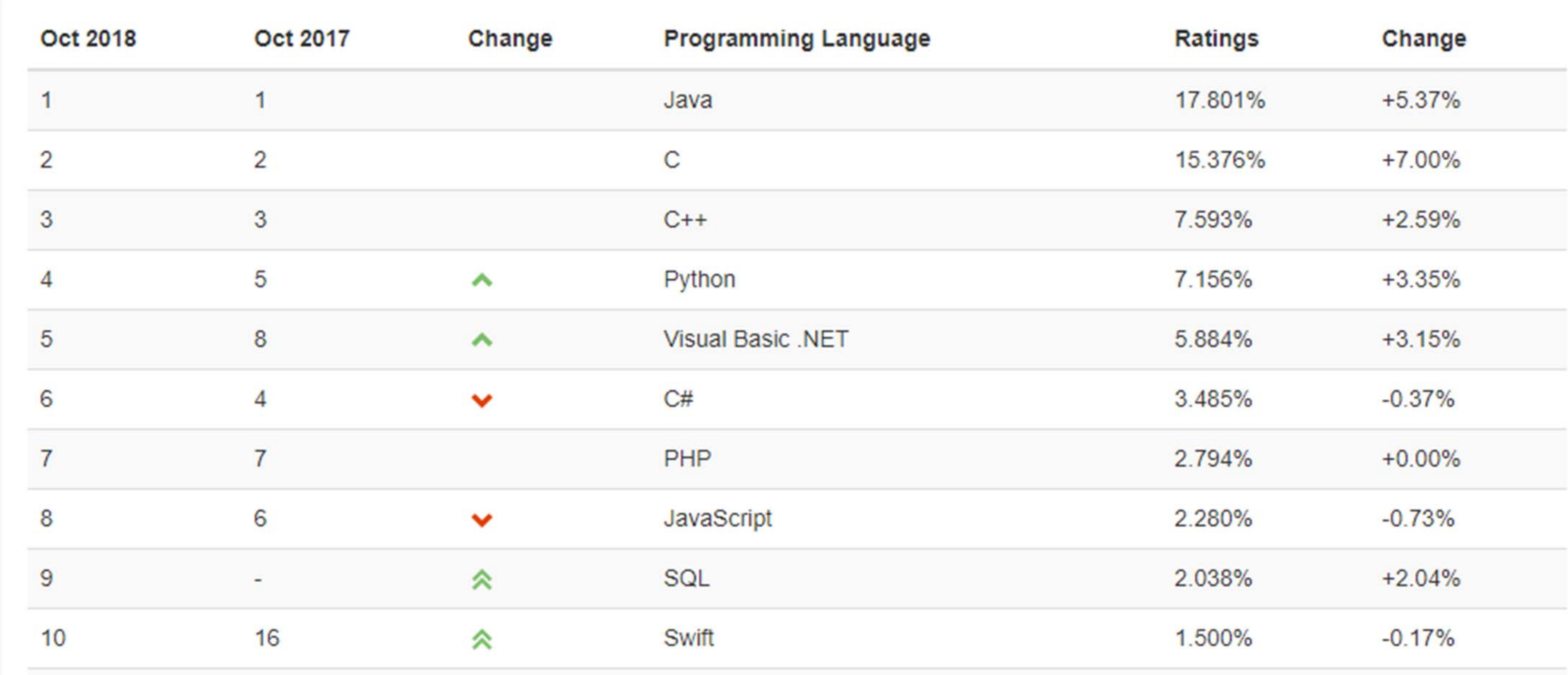
Why Python, not others?

- ➢ Interpreter-based open source script language
  - ✓ Efficiency
    - efficient for programmers to write programs
    - but Python programs themselves are not efficient
  - ✓ Memory safety
  - ✓ Thread safety
  - ✓ Easier to learn

Why Python, not others?

- ➢ Multiply Programming Paradigms
  - Imperative —— How to do
    - procedural which groups instructions into procedures, e.g., C
    - object-oriented which groups instructions together with the part of the state they operate on, e.g., C++
  - Functional —— What to do
    - Functions as first-class objects and data collections, e.g., Haskell

Why Python, not others?



Oct 2018	Oct 2017	Change	Programming Language	Ratings	Change
1	1		Java	17.801%	+5.37%
2	2		C	15.376%	+7.00%
3	3		C++	7.593%	+2.59%
4	5	▲	Python	7.156%	+3.35%
5	8	▲	Visual Basic .NET	5.884%	+3.15%
6	4	▼	C#	3.485%	-0.37%
7	7		PHP	2.794%	+0.00%
8	6	▼	JavaScript	2.280%	-0.73%
9	-	▲▲	SQL	2.038%	+2.04%
10	16	▲▲	Swift	1.500%	-0.17%

https://www.tiobe.com/tiobe-index/

Does anyone really use Python?

So how do I get started?

Step 1: Python Environment

- Install development environment:
  - ➢ Download python 3.X at www.python.org/downloads
  - ➢ 32bit/64bit, Windows, Linux/UNIX, Mac OSX depends on your computer
  - ➢ Run python.exe or idle.bat in Windows
  - ➢ Typing python3 or python on Linux or Mac OSX
  - ➢ Two programming modes in Python
    - Interactive
    - Batch

How do I know I installed everything correctly?

Hello World in C

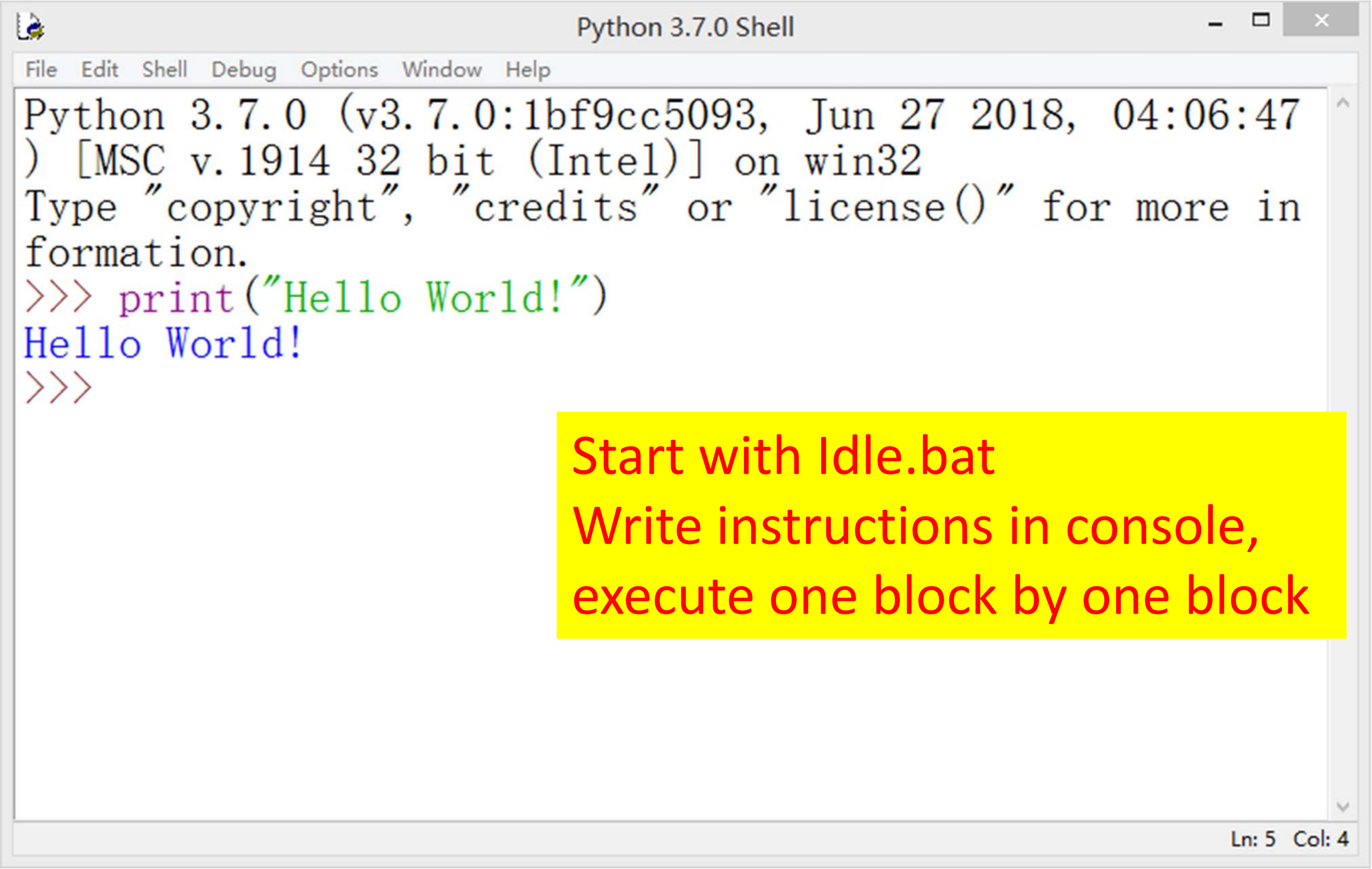
```c
//hello.c
#include <stdio.h>
int main()
{
    printf("Hello World\n");
    return 0;
}
```

Compilation & run

gcc hello.c –o hello (create hello.exe)

./hello.exe (run it)

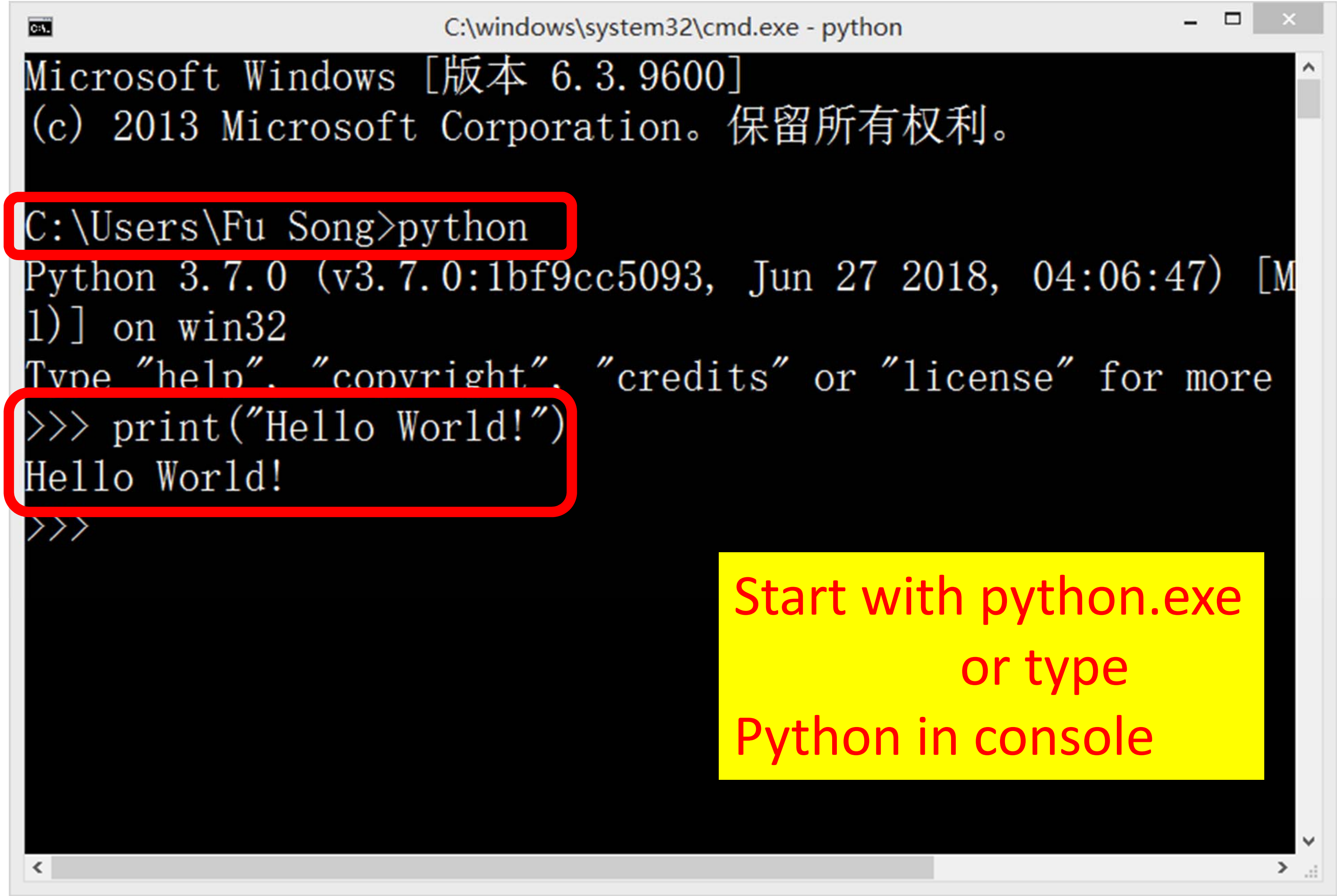
Step 2: Interactive programming



```
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:06:47
) [MSC v.1914 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more in
formation.
>>> print("Hello World!")
Hello World!
>>>
```

Start with Idle.bat
Write instructions in console,
execute one block by one block

Step 2: Interactive programming

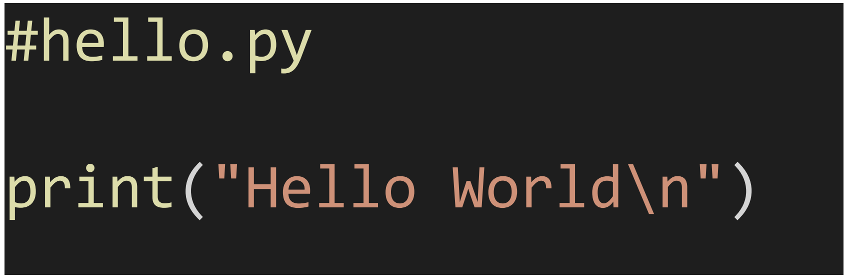


```
Microsoft Windows [版本 6.3.9600]
(c) 2013 Microsoft Corporation。保留所有权利。

C:\Users\Fu Song>python
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:06:47) [M
1)] on win32
Type "help", "copyright", "credits" or "license" for more
>>> print("Hello World!")
Hello World!
>>>
```

Start with python.exe
or type
Python in console

Step 2: Batch programming

Comment starts with #



```
#hello.py

print("Hello World\n")
```

Write instructions in a file hello.py, execute the file by

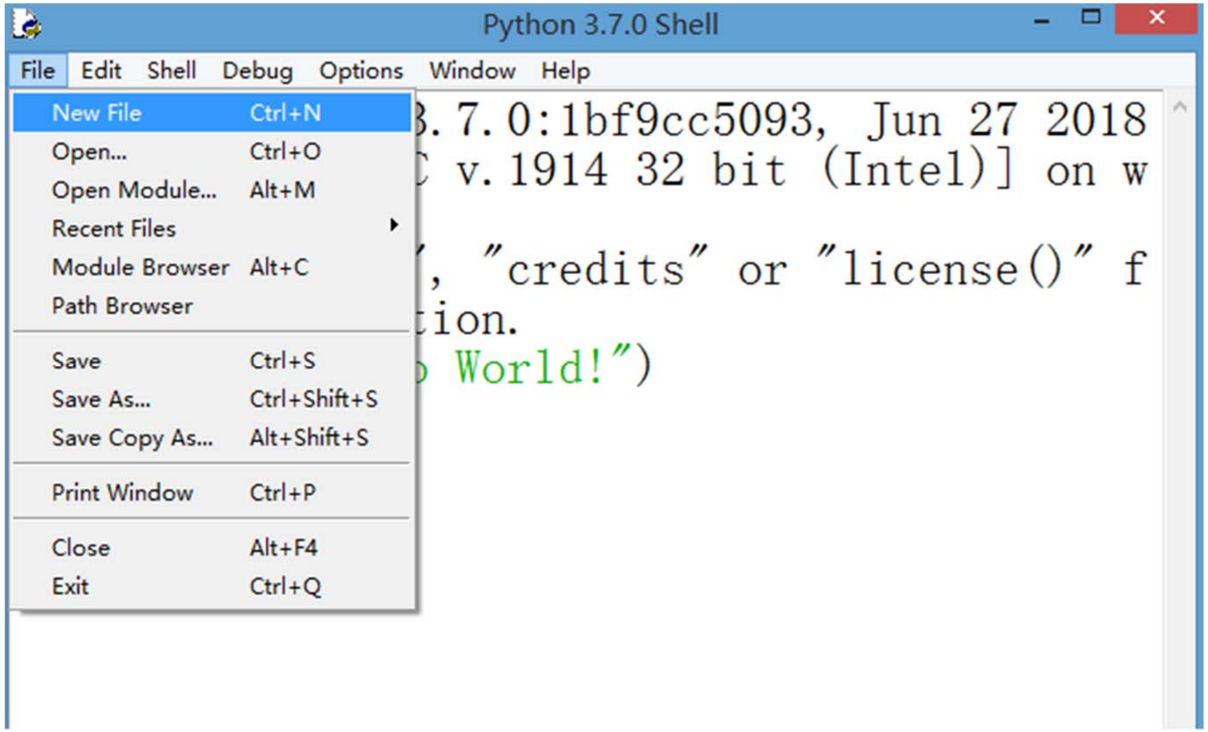
python hello.py

No executable file is generated

Step 2: Batch programming

In IDLE:

1. “File”==>“New File” create a file,
2. write your python program
3. save file as filename.py



```
3.7.0:1bf9cc5093, Jun 27 2018
C v.1914 32 bit (Intel)] on w

", "credits" or "license()" f
tion.
o World!")
```

Step 2: Batch programming

In IDLE:

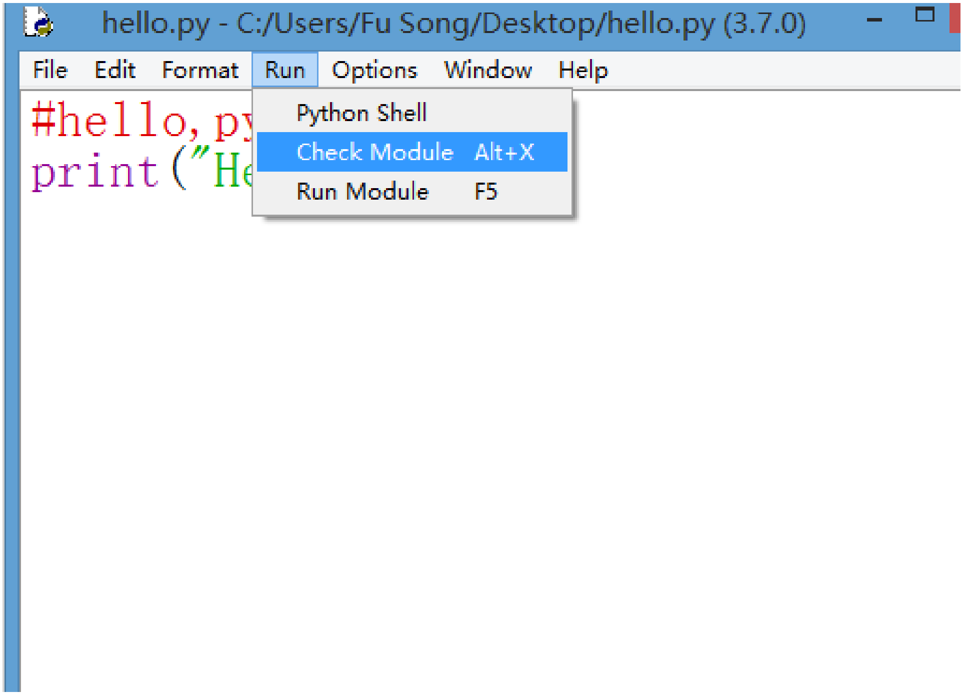
1. “File”==>“New File” create a file,
2. write your python program
3. save file as filename.py
4. “Run”==>”Check Module”

   check syntax

5. “Run”==>”Run Module”

   run the program and

   print the result in IDLE



```
#hello, py
print("He
```

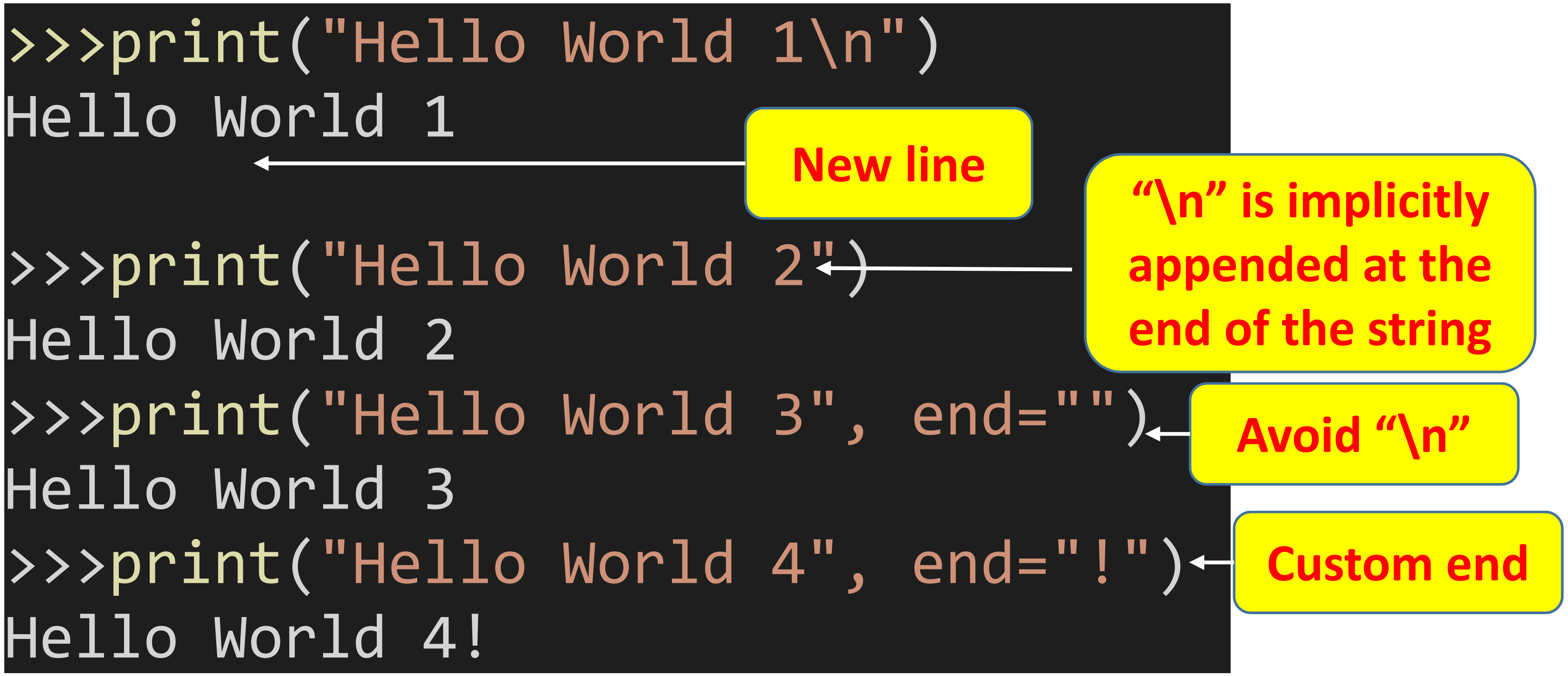
Simple IO in Python

Print in Python

```
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)
```

- Print objects to the text stream file, separated by sep and followed by end
- All objects are converted to strings like str() does and written to the file
- Both sep and end must be strings; they can also be None, which means to use the default values
- If no objects are given, print() will just write end
- The file argument must be an object with a write(string) method; sys.stdout is the default stream
- Whether output is buffered is usually determined by file, but if the flush keyword argument is true, the stream is forcibly flushed

More Hello World in Python



```
>>>print("Hello World 1\n")
Hello World 1

>>>print("Hello World 2")
Hello World 2
>>>print("Hello World 3", end="")
Hello World 3
>>>print("Hello World 4", end="!")
Hello World 4!
```

New line

“\n” is implicitly appended at the end of the string

Avoid “\n”

Custom end

More Hello World in Python

```python
#hello.py
print("Hello World 1")
print("Hello World 2",end="")
print("Hello World 3",end="!")
print("Hello World 4",end="\n")
print("Hello World 5")
```

Output

```
Hello World 1
Hello World 2Hello World 3!Hello World 4
Hello World 5
>>>
```

More Hello World in Python

```
#hello.py
print("Hello World")
print("Hello","World",sep="&",end="!\n")
print("Hello","World", sep="-",end="!")
```

Output

```
Hello World
Hello&World!
Hello-World!
>>>
```

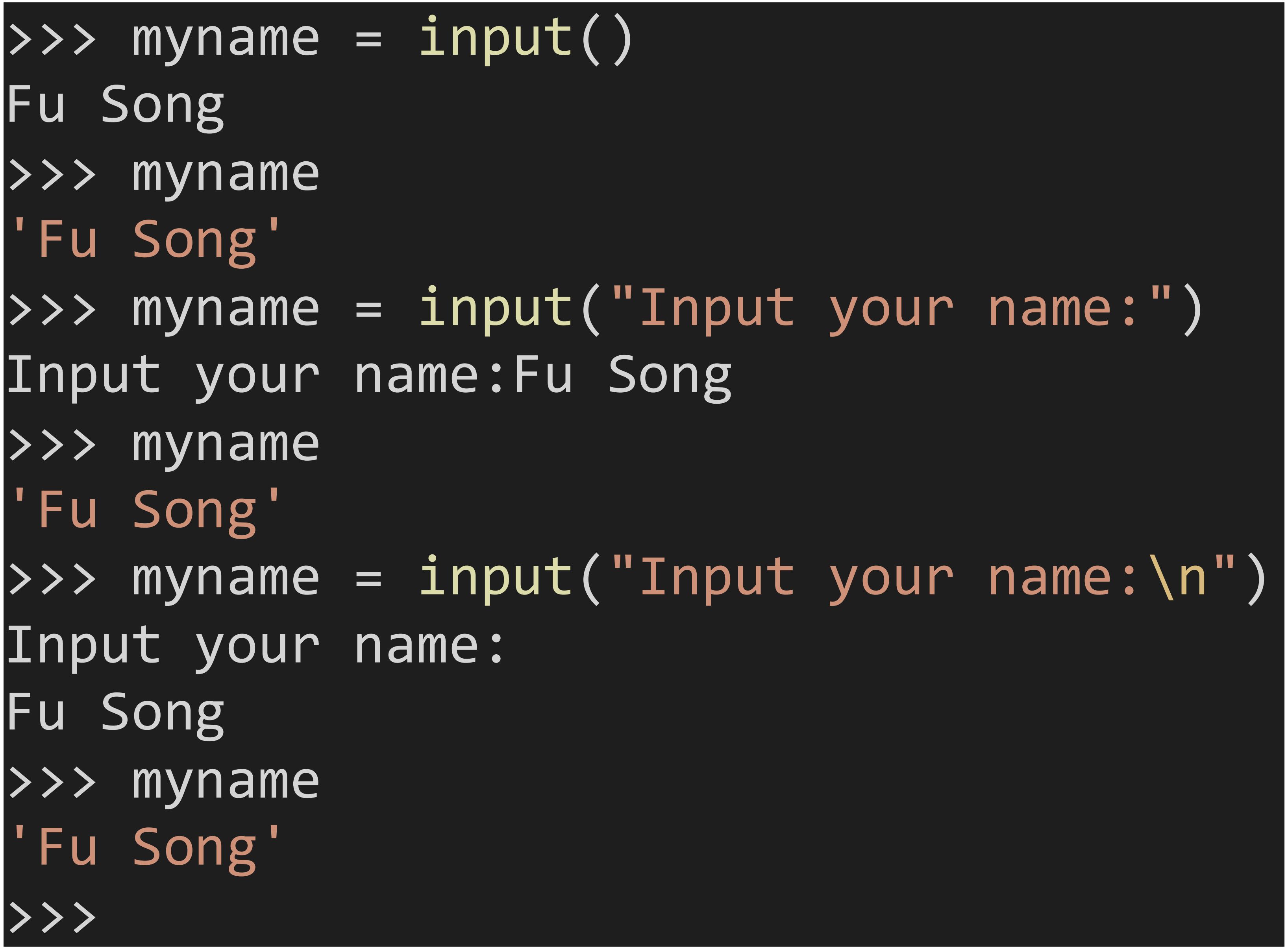
Input in Python

input([prompt])

- **If the prompt argument is present, it is written to standard output without a trailing newline**

- **The function then reads a line from input, converts it to a string (stripping a trailing newline), and returns that. When EOF is read, EOFError is raised**

input([prompt])



```
>>> myname = input()
Fu Song
>>> myname
'Fu Song'
>>> myname = input("Input your name:")
Input your name:Fu Song
>>> myname
'Fu Song'
>>> myname = input("Input your name:\n")
Input your name:
Fu Song
>>> myname
'Fu Song'
>>>
```

C/C++ vs Python

C/C++ vs Python

- C/C++
  - Procedural + OO
  - Compilation
  - Bounded int/float
  - Weakly and statically typed language

- Python
  - Procedural + OO (pure OO, every value is an object)
  - Interpretation
  - Unbounded int/float
  - Strongly and dynamically typed language

C/C++ vs Python

- C/C++
  - Procedural + OO
  - Compilation
  - Bounded int/float
  - Weakly and statically typed language

- Python
  - Procedural + OO (pure OO, every value is an object)
  - Interpretation
  - Unbounded int/float
  - Strongly and dynamically typed language

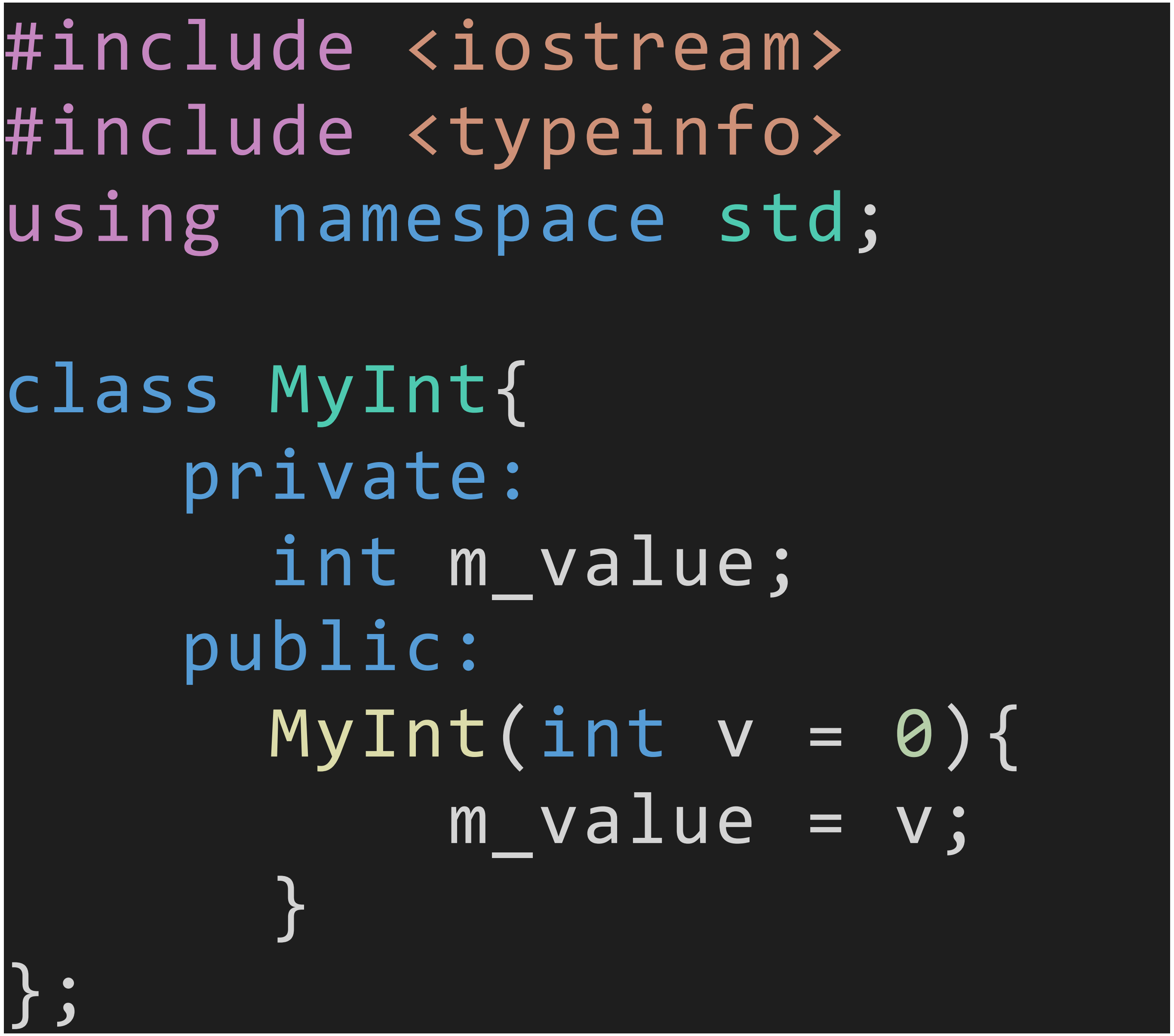
OO in C++

Defined in header <typeinfo>
class type_info;

typeid(type)	returns the type t
typeid(value)	returns the type t of the value
t.name()	returns the name of the type t

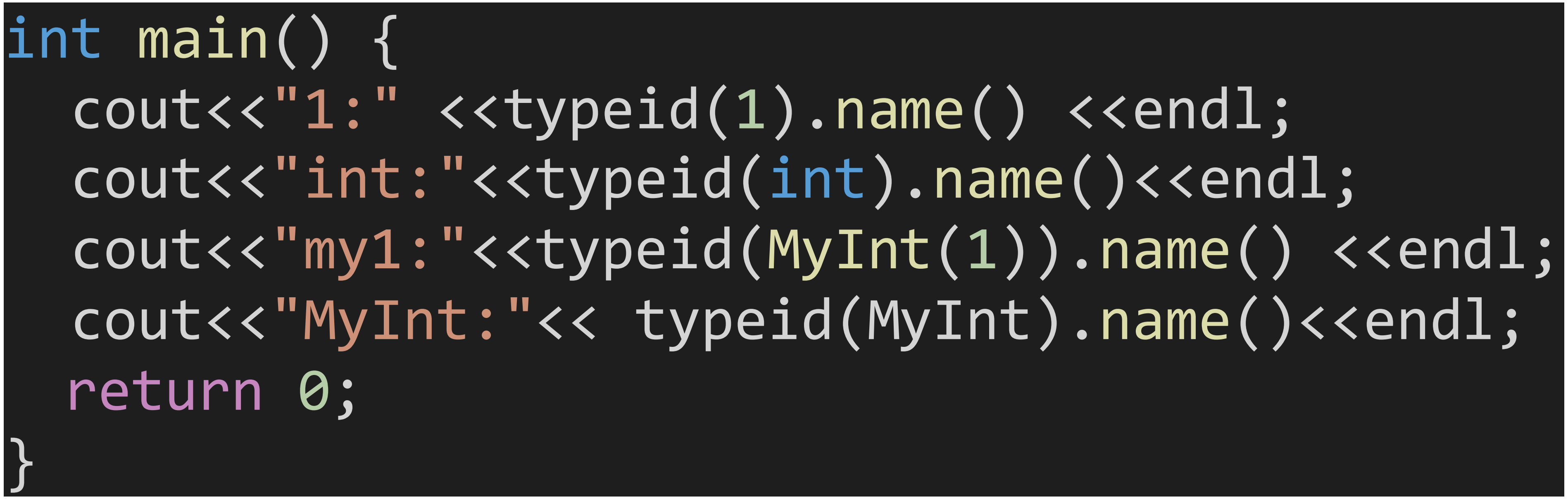
The name of the type t may differ for different compilers

OO in C++



```
#include <iostream>
#include <typeinfo>
using namespace std;

class MyInt{
    private:
      int m_value;
    public:
      MyInt(int v = 0){
          m_value = v;
      }
};
```

OO in C++

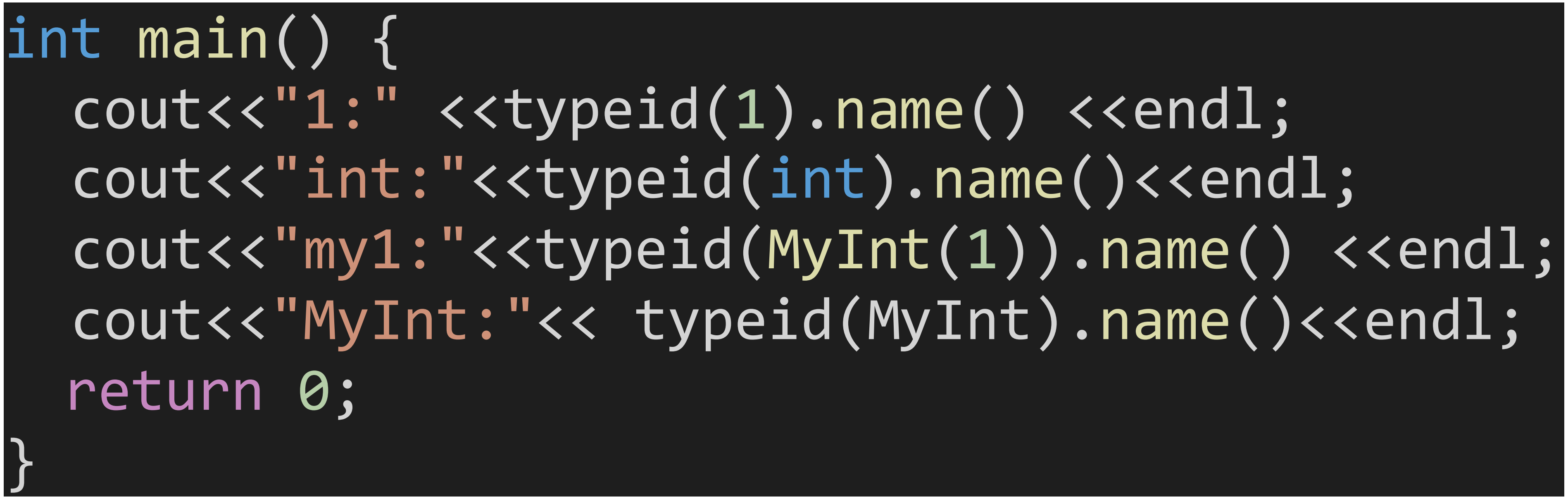


```cpp
int main() {
  cout<<"1:" <<typeid(1).name() <<endl;
  cout<<"int:"<<typeid(int).name()<<endl;
  cout<<"my1:"<<typeid(MyInt(1)).name() <<endl;
  cout<<"MyInt:"<< typeid(MyInt).name()<<endl;
  return 0;
}
```

Output in VC++
19.00.23506 for x64

```
Type of 1: int
Type of int: int
Type of my1: class MyInt
Type of MyInt: class MyInt
```

OO in C++



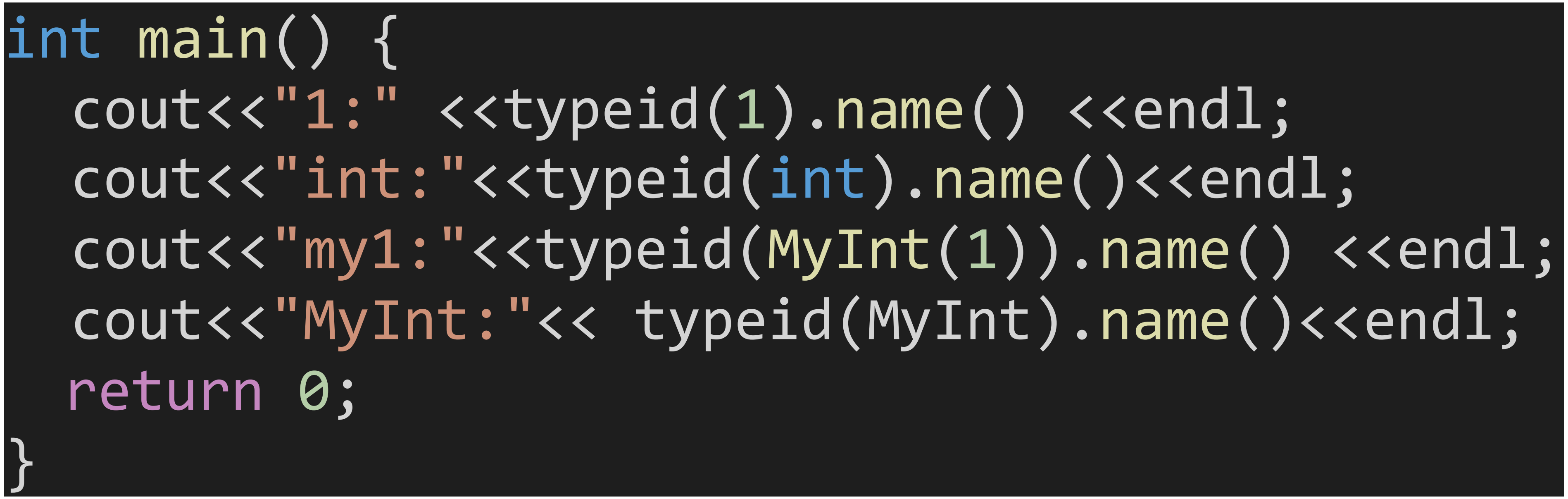
```
int main() {
   cout<<"1:" <<typeid(1).name() <<endl;
   cout<<"int:"<<typeid(int).name()<<endl;
   cout<<"my1:"<<typeid(MyInt(1)).name() <<endl;
   cout<<"MyInt:"<< typeid(MyInt).name()<<endl;
  return 0;
}
```

Output in clang 3.8.0

```
Type of 1: i
Type of int: i
Type of my1: 5MyInt
Type of MyInt: 5MyInt
```

5 is the length of the name

OO in C++



```
int main() {
  cout<<"1:" <<typeid(1).name() <<endl;
  cout<<"int:"<<typeid(int).name()<<endl;
  cout<<"my1:"<<typeid(MyInt(1)).name() <<endl;
  cout<<"MyInt:"<< typeid(MyInt).name()<<endl;
  return 0;
}
```

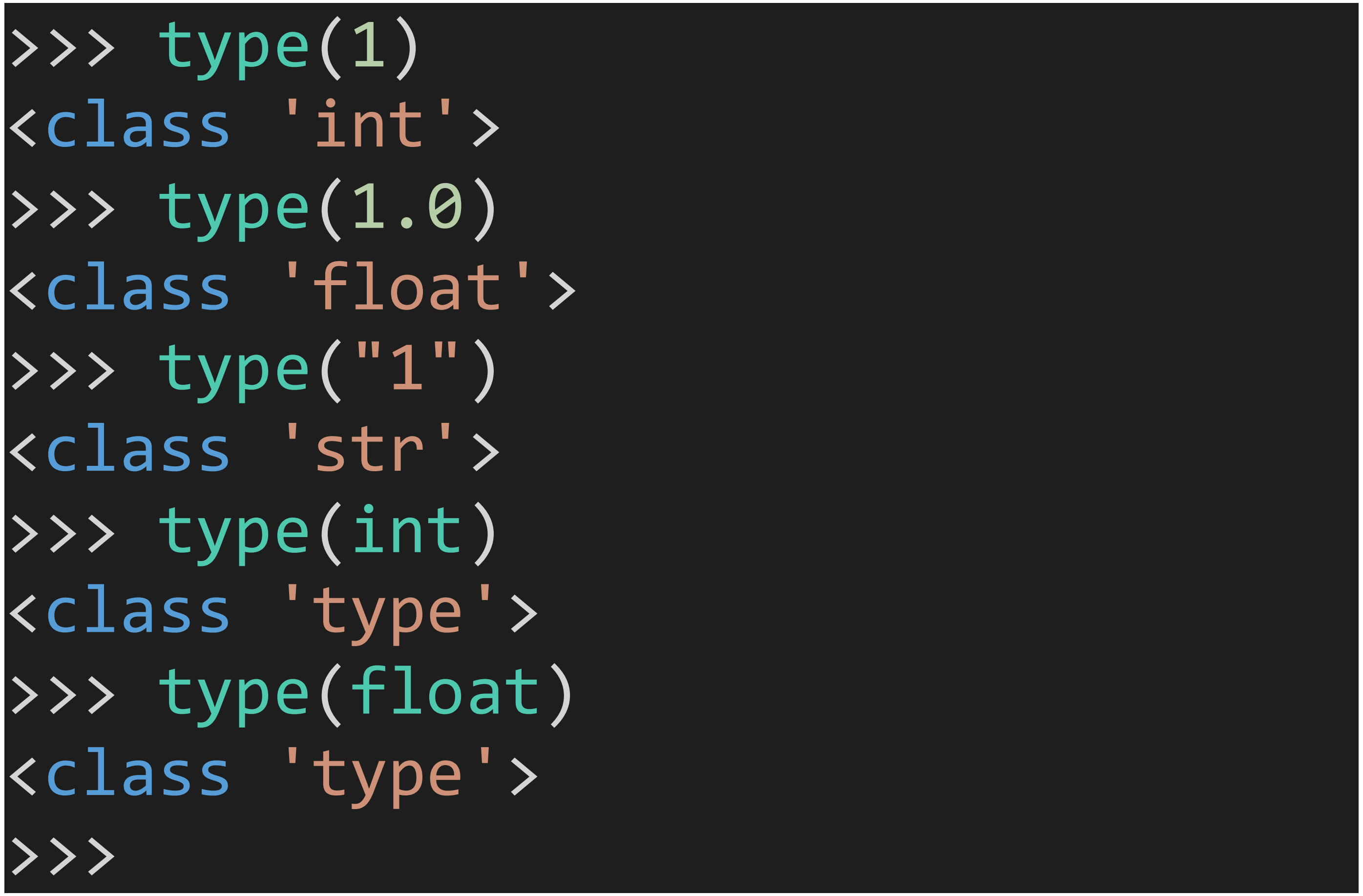
Output in g++ 5.4.0

```
Type of 1: i
Type of int: i
Type of my1: 5MyInt
Type of MyInt: 5MyInt
```

5 is the length of the name

Pure OO in Python

type(object) returns the type of object



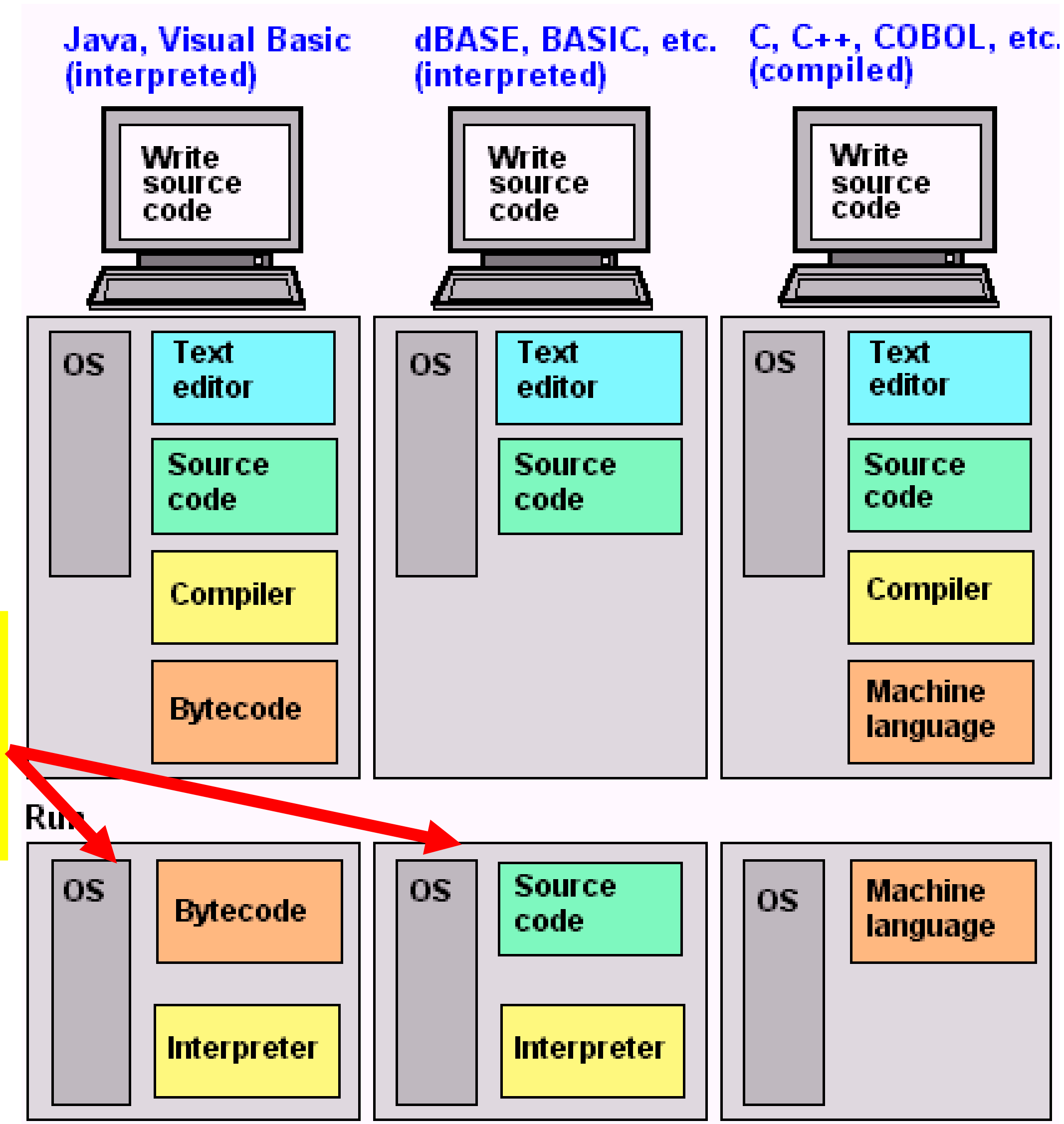
```
>>> type(1)
<class 'int'>
>>> type(1.0)
<class 'float'>
>>> type("1")
<class 'str'>
>>> type(int)
<class 'type'>
>>> type(float)
<class 'type'>
>>>
```

C/C++ vs Python

- C/C++
  - Procedural + OO
  - Compilation
  - Bounded int/float
  - Weakly and statically typed language

- Python
  - Procedural + OO (pure OO, every value is an object)
  - Interpretation
  - Unbounded int/float
  - Strongly and dynamically typed language

Compilation vs Interpretation



Java, Visual Basic (interpreted)

Write source code

OS

Text editor

Source code

Compiler

Bytecode

dBASE, BASIC, etc. (interpreted)

Write source code

OS

Text editor

Source code

C, C++, COBOL, etc. (compiled)

Write source code

OS

Text editor

Source code

Compiler

Machine language

Run

OS

Bytecode

Interpreter

OS

Source code

Interpreter

OS

Machine language

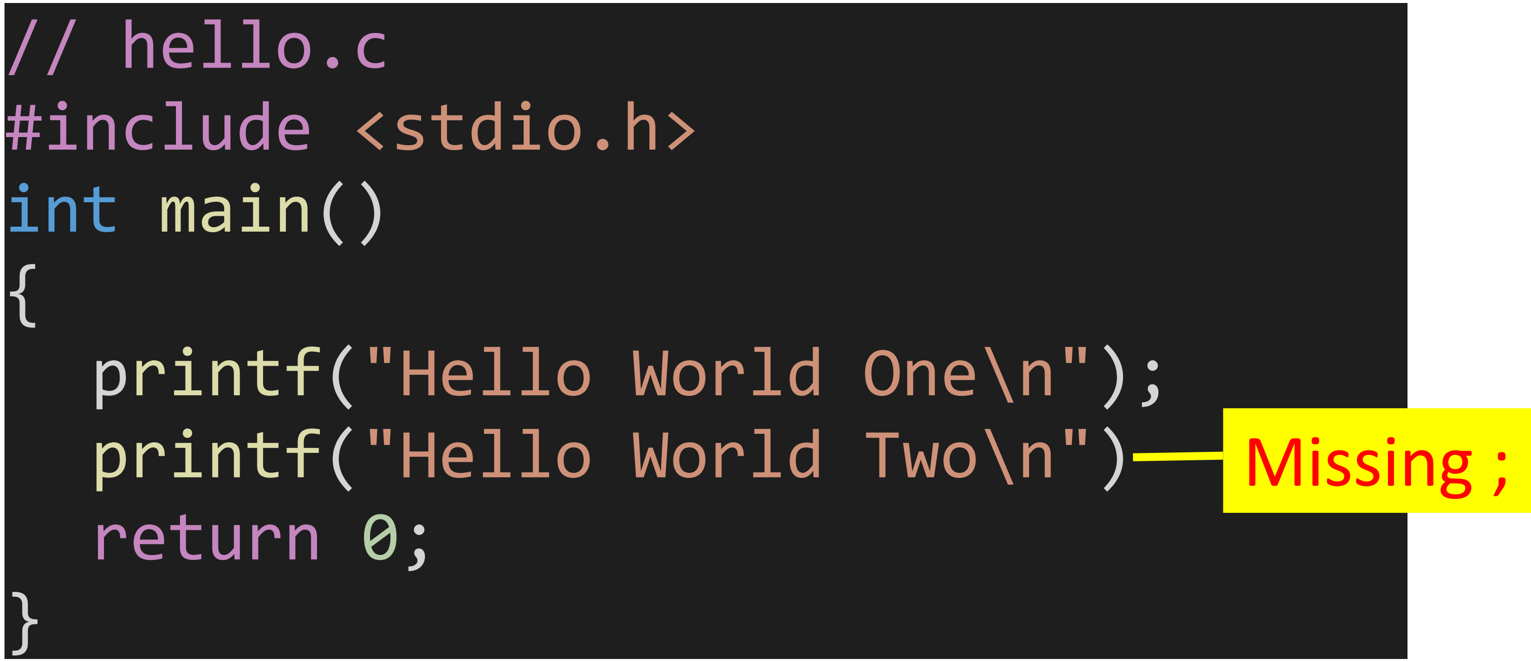
Python supports both interpretation modes

Compilation vs Interpretation

No	Compiler	Interpreter
1	Takes **Entire** program as input	Takes **Single** instruction as input
2	Intermediate object code is **Generated**	Intermediate object code may **not** be **Generated**
3	Execute **faster**	Execute **slower**
4	**Memory requirement** : **More** (Since Object Code is Generated)	**Memory Requirement** is **Less**
5	Program need not be **compiled** every time	Every time higher level program is converted into lower level program
6	**Errors** are displayed after **entire program** is checked	**Errors** are displayed for **every instruction** interpreted (if any)
7	**Example** : C/C++ Compiler	**Example** : Python

Entire vs Single

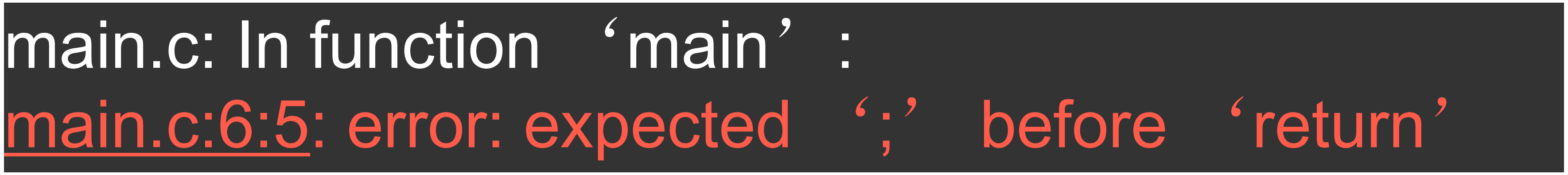
C program



```c
// hello.c
#include <stdio.h>
int main()
{
   printf("Hello World One\n");
   printf("Hello World Two\n")
   return 0;
}
```

Missing ;

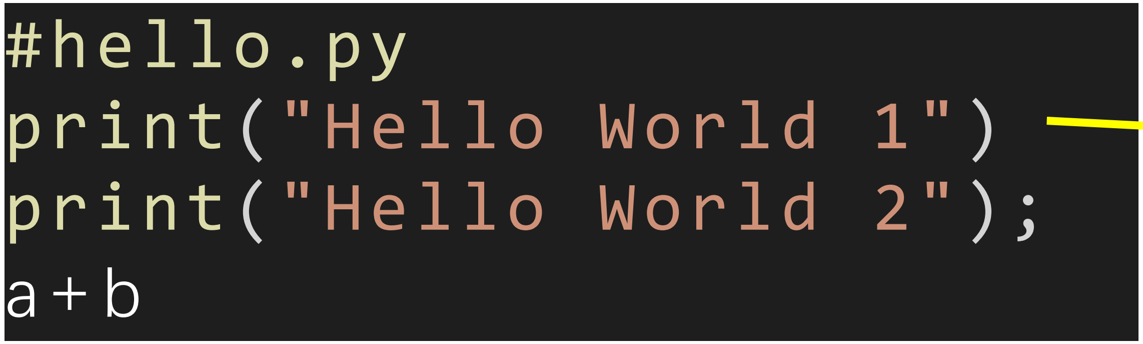
gcc hello.c –o hello



```
main.c: In function ‘main’ :
main.c:6:5: error: expected ‘;’ before ‘return’
```

Entire vs Single

Python program



```
#hello.py
print("Hello World 1")
print("Hello World 2");
a+b
```

Missing ; ?

Output

```
Hello World 1
Hello World 2
Traceback (most recent call last):
  ... line 3, in <module>
    a+b
NameError: name 'a' is not defined
```

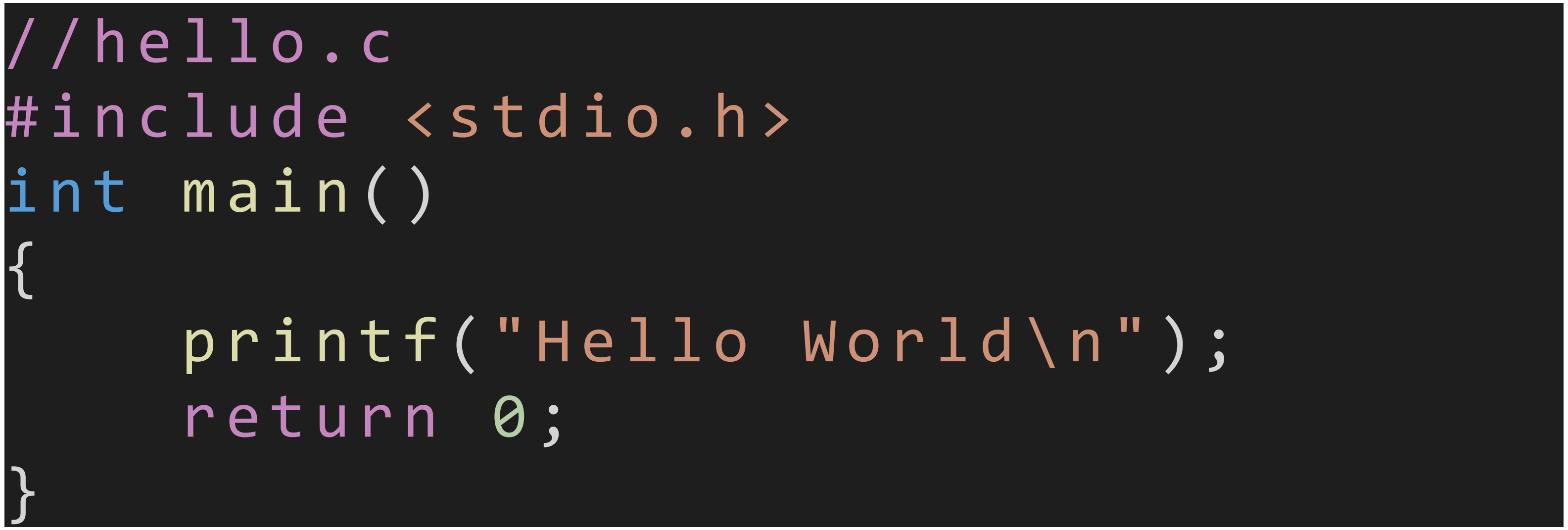
print(..) were executed

Compilation vs Interpretation

No	Compiler	Interpreter
1	Takes **Entire** program as input	Takes **Single** instruction as input
2	Intermediate object code is **Generated**	Intermediate object code may **not** be **Generated**
3	Execute **faster**	Execute **slower**
4	**Memory requirement** : **More** (Since Object Code is Generated)	**Memory Requirement** is **Less**
5	Program need not be **compiled** every time	Every time higher level program is converted into lower level program
6	**Errors** are displayed after **entire program** is checked	**Errors** are displayed for **every instruction** interpreted (if any)
7	**Example** : C/C++ Compiler	**Example** : Python

Intermediate object code is Generated

C program



```c
//hello.c
#include <stdio.h>
int main()
{
    printf("Hello World\n");
    return 0;
}
```

Compilation & run

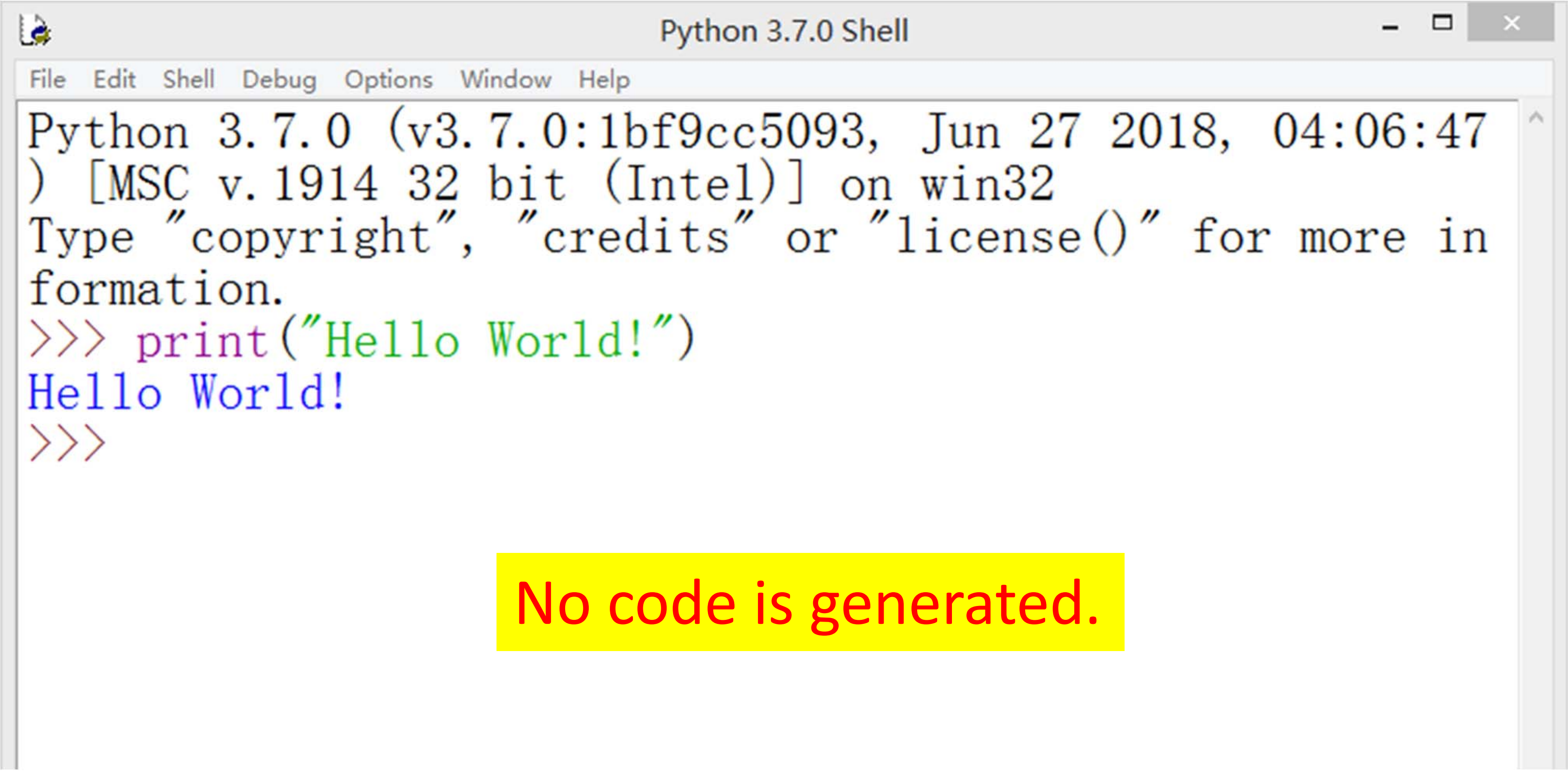
gcc hello.c –o hello (create **hello.exe**)

./hello.exe (run it)

- Executable code hello.exe **depends** on OS and CPU

Intermediate object code is NOT Generated

Python program (source code)

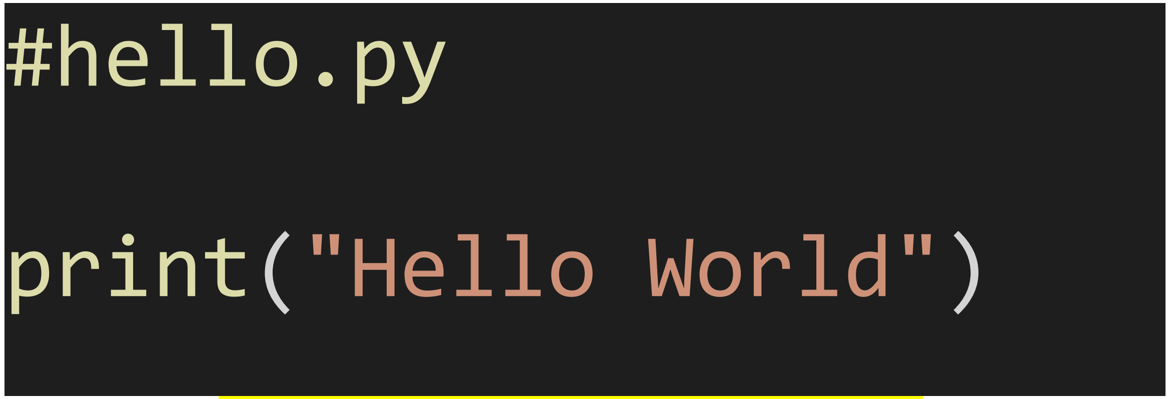


```
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:06:47
) [MSC v.1914 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more in
formation.
>>> print("Hello World!")
Hello World!
>>>
```

No code is generated.

Intermediate object code is NOT Generated

Python program (source code)



```
#hello.py

print("Hello World")
```

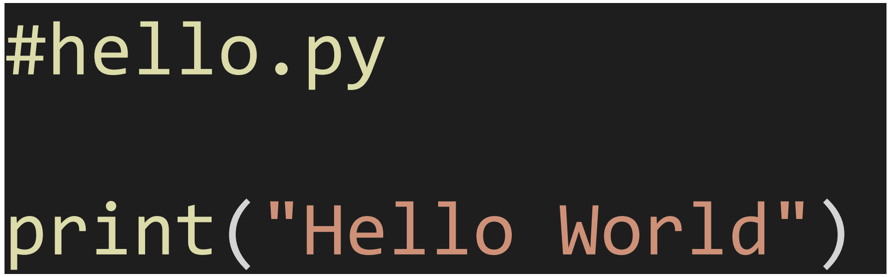
No code is generated.

Run (without compilation)

```
python hello.py
```

Intermediate object code is Generated

Python program (bytecode code)



```
#hello.py

print("Hello World")
```

Compilation

Python –m py_compile hello.py

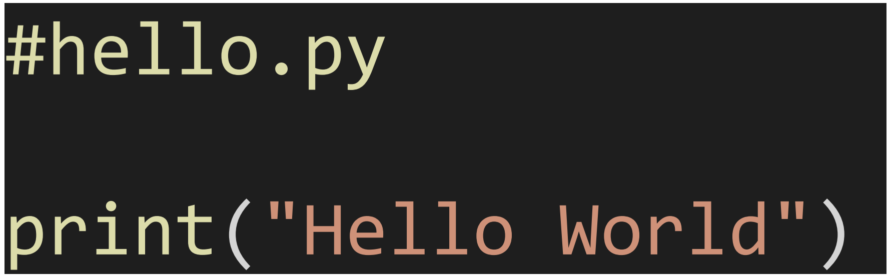
Generate hello.cpython-37.pyc in __pycache__ directory

37 is version number, it depends on your system.

- Bytecode **does not depend** on OS and CPU

Intermediate object code is Generated

Python program (bytecode code)



```
#hello.py

print("Hello World")
```

Compilation

```
Python –m py_compile hello.py
```

Interpretation

```
Python  __pycache__/printhello.cpython-37.pyc
```

Output

```
Hello World
```

Insights

When run a file.py
the interpreter generates PyCodeObject in memory
and executes PyCodeObject
After termination, PyCodeObject is deleted in memory

Python –m py_compile file.py
generates PyCodeObject in memory and save it in disk

When run a file.pyc
the interpreter loads PyCodeObject into memory
from disk and executes file.pyc
After termination, file.pyc is still in disk

The latter is more efficient

Compilation Multiple Files

Compile several files

```
Python –m py_compile file1.py file2.py …
```

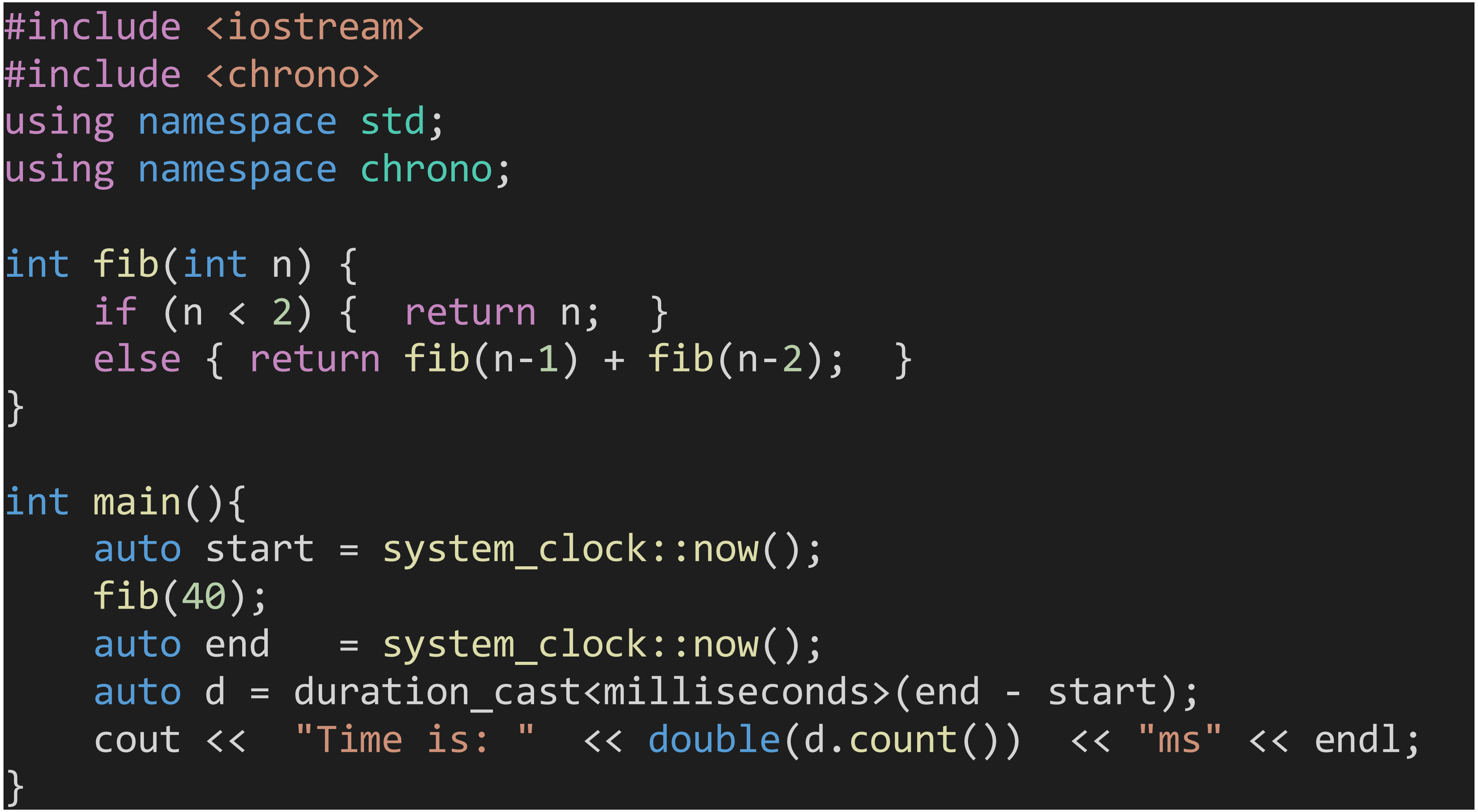
Compile all files in directory /cs100/

```
Python –m py_compile compileall   /cs100/
```

Compilation vs Interpretation

No	Compiler	Interpreter
1	Takes **Entire** program as input	Takes **Single** instruction as input
2	Intermediate object code is **Generated**	Intermediate object code may **not** be **Generated**
3	Execute **faster**	Execute **slower**
4	**Memory requirement** : **More** (Since Object Code is Generated)	**Memory Requirement** is **Less**
5	Program need not be **compiled** every time	Every time higher level program is converted into lower level program
6	**Errors** are displayed after **entire program** is checked	**Errors** are displayed for **every instruction** interpreted (if any)
7	**Example** : C/C++ Compiler	**Example** : Python

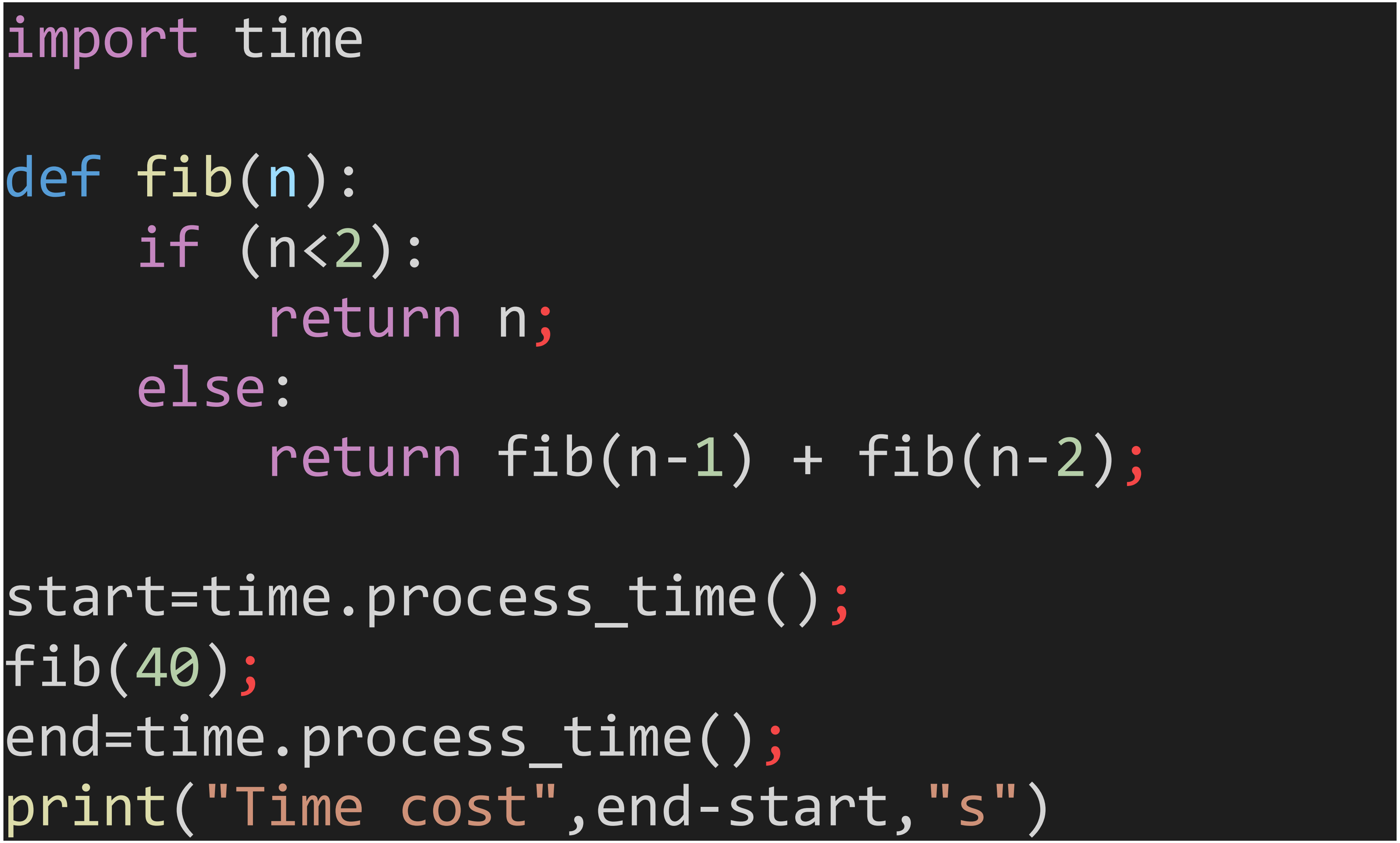
Measure times in C/C++



```cpp
#include <iostream>
#include <chrono>
using namespace std;
using namespace chrono;

int fib(int n) {
    if (n < 2) {  return n;  }
    else { return fib(n-1) + fib(n-2);  }
}

int main(){
    auto start = system_clock::now();
    fib(40);
    auto end   = system_clock::now();
    auto d = duration_cast<milliseconds>(end - start);
    cout <<  "Time is: "  << double(d.count())  << "ms" << endl;
}
```

Time is: 8388ms

Measure times in Python



```python
import time

def fib(n):
    if (n<2):
        return n;
    else:
        return fib(n-1) + fib(n-2);

start=time.process_time();
fib(40);
end=time.process_time();
print("Time cost",end-start,"s")
```

Time cost 67.0625 s

C/C++ vs Python

- C/C++
  - Procedural + OO
  - Compilation
  - Bounded number
  - Weakly and statically typed language

- Python
  - Procedural + OO (pure OO, every value is an object)
  - Interpretation
  - Unbounded number
  - Strongly and dynamically typed language

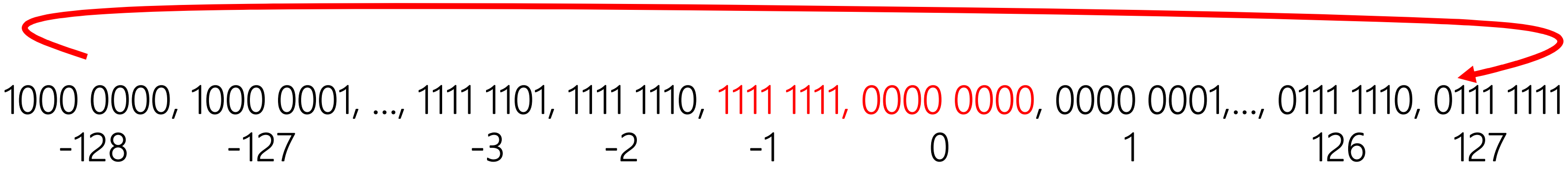
Number in C/C++

Integers in C are
(depending on os and cpu)

- short (2 bytes = 16 bits)
- int (2 bytes)
- long (4 bytes = 32 bits)
- unsigned (2 bytes)
- unsigned short (2 bytes)
- unsigned long 32 bits (4 bytes)

signed type with n bits:
range is -2^{n-1} -- 2^{n-1}-1

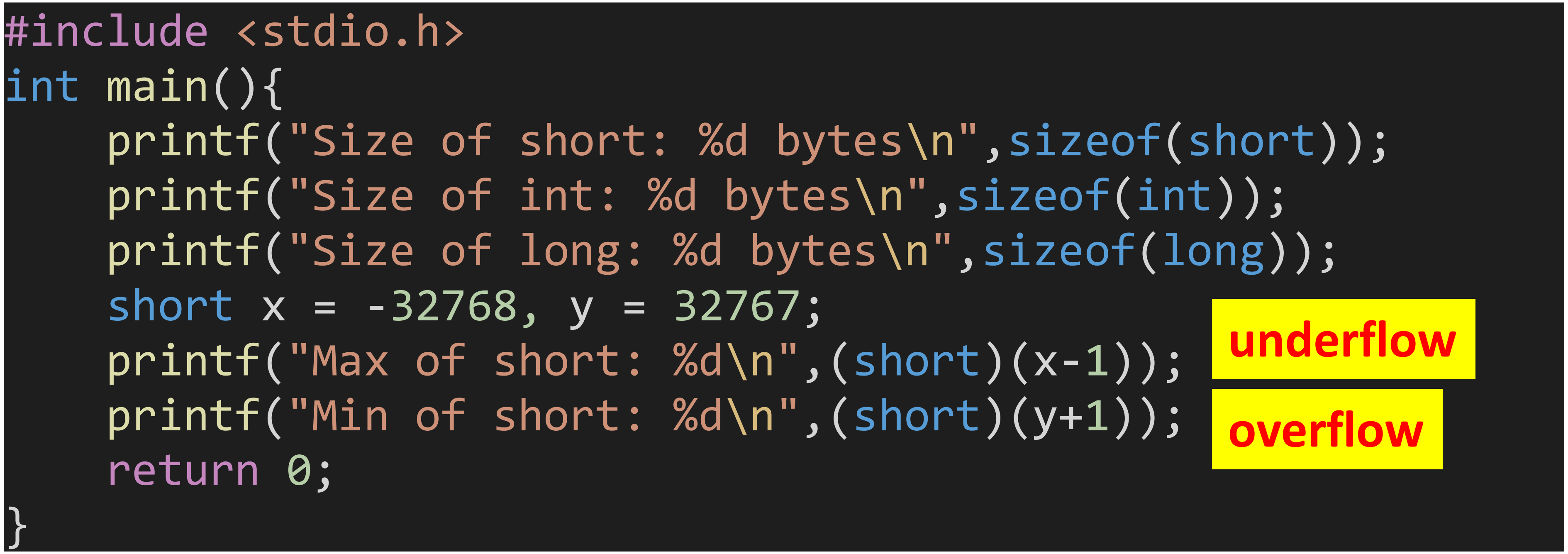
unsigned type with n bits:
range is 0 -- 2^{n} -1



1000 0000,	1000 0001,	...,	1111 1101,	1111 1110,	1111 1111,	0000 0000,	0000 0001,	...,	0111 1110,	0111 1111
-128	-127		-3	-2	-1	0	1		126	127

The case of signed type with 8 bits

Number in C/C++



```c
#include <stdio.h>
int main(){
    printf("Size of short: %d bytes\n",sizeof(short));
    printf("Size of int: %d bytes\n",sizeof(int));
    printf("Size of long: %d bytes\n",sizeof(long));
    short x = -32768, y = 32767;
    printf("Max of short: %d\n",(short)(x-1));
    printf("Min of short: %d\n",(short)(y+1));
    return 0;
}
```

underflow

overflow

Output

```
Size of short: 2 bytes
Size of int: 4 bytes
Size of long: 8 bytes
Max of short: 32767
Min of short: -32768
```

-32768 = 1000 0000 0000 0000

32767= 0111 1111 1111 1111

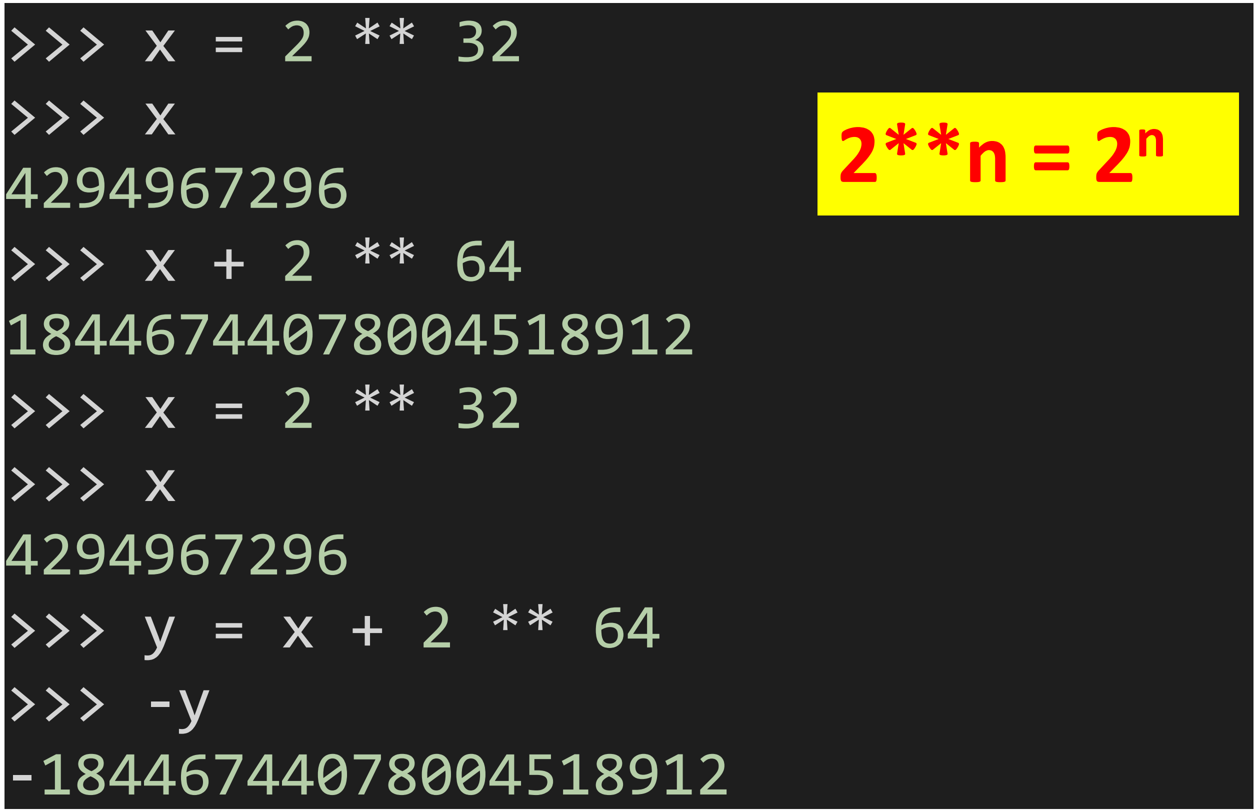
1000 0000 0000 0000 - 1 = 0111 1111 1111 1111

0111 1111 1111 1111 +1 = 1000 0000 0000 0000

Number in Python

In python3 (not python2),

- numbers (integer and float) are **unbounded**



```
>>> x = 2 ** 32
>>> x
4294967296
>>> x + 2 ** 64
18446744078004518912
>>> x = 2 ** 32
>>> x
4294967296
>>> y = x + 2 ** 64
>>> -y
-18446744078004518912
```

2n = 2^n**

C/C++ vs Python

- C/C++
  - Procedural + OO
  - Compilation
  - Bounded number
  - Weakly and statically typed language

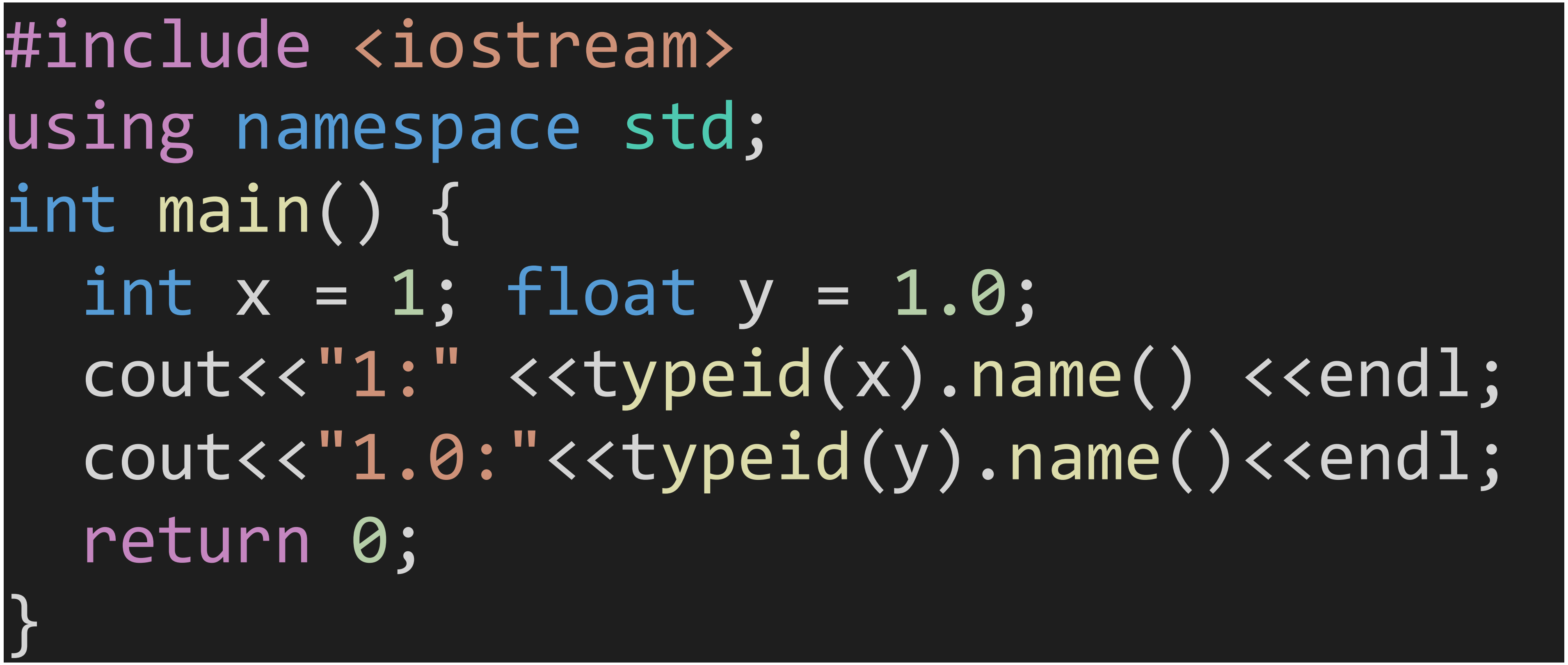
- Python
  - Procedural + OO (pure OO, every value is an object)
  - Interpretation
  - Unbounded number
  - Strongly and dynamically typed language

Static typing vs Dynamic typing

- Static typing (C/C++)
  - Types are determined by programs and checking by compiler without executing them
  - Each variable has same type in different executions

- Dynamic typing (Python)
  - Types are determined during execution
  - Each variable can have different types in different executions, even at different time points in the same execution

C/C++: Static Typing

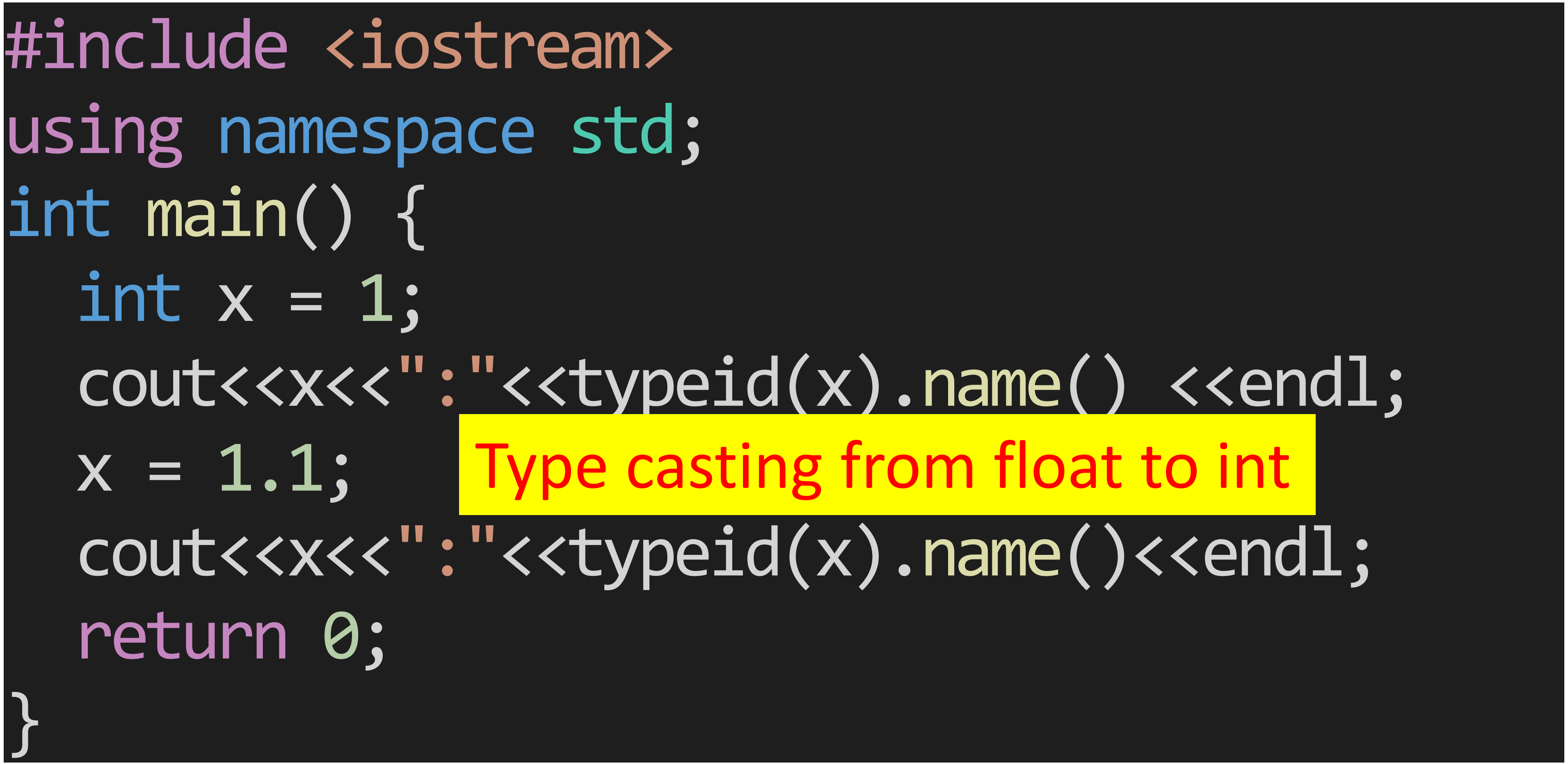


```
#include <iostream>
using namespace std;
int main() {
  int x = 1; float y = 1.0;
  cout<<"1:" <<typeid(x).name() <<endl;
  cout<<"1.0:"<<typeid(y).name()<<endl;
  return 0;
}
```

Output in VC++
19.00.23506 for x64

```
1: int
1.0: float
```

C/C++: Static Typing



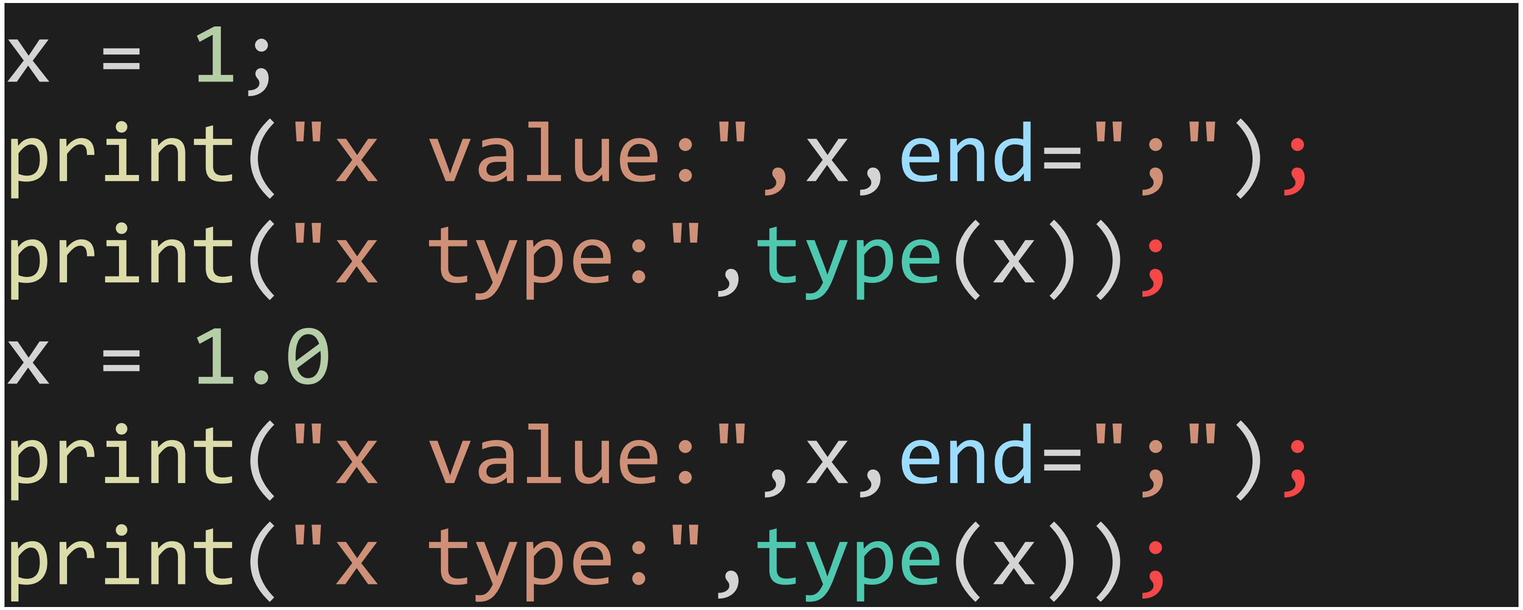
```cpp
#include <iostream>
using namespace std;
int main() {
  int x = 1;
  cout<<x<<":"<<typeid(x).name() <<endl;
  x = 1.1;
  cout<<x<<":"<<typeid(x).name()<<endl;
  return 0;
}
```

Type casting from float to int

Output in VC++
19.00.23506 for x64

```
1: int
1: int
```

Python: Dynamic Typing



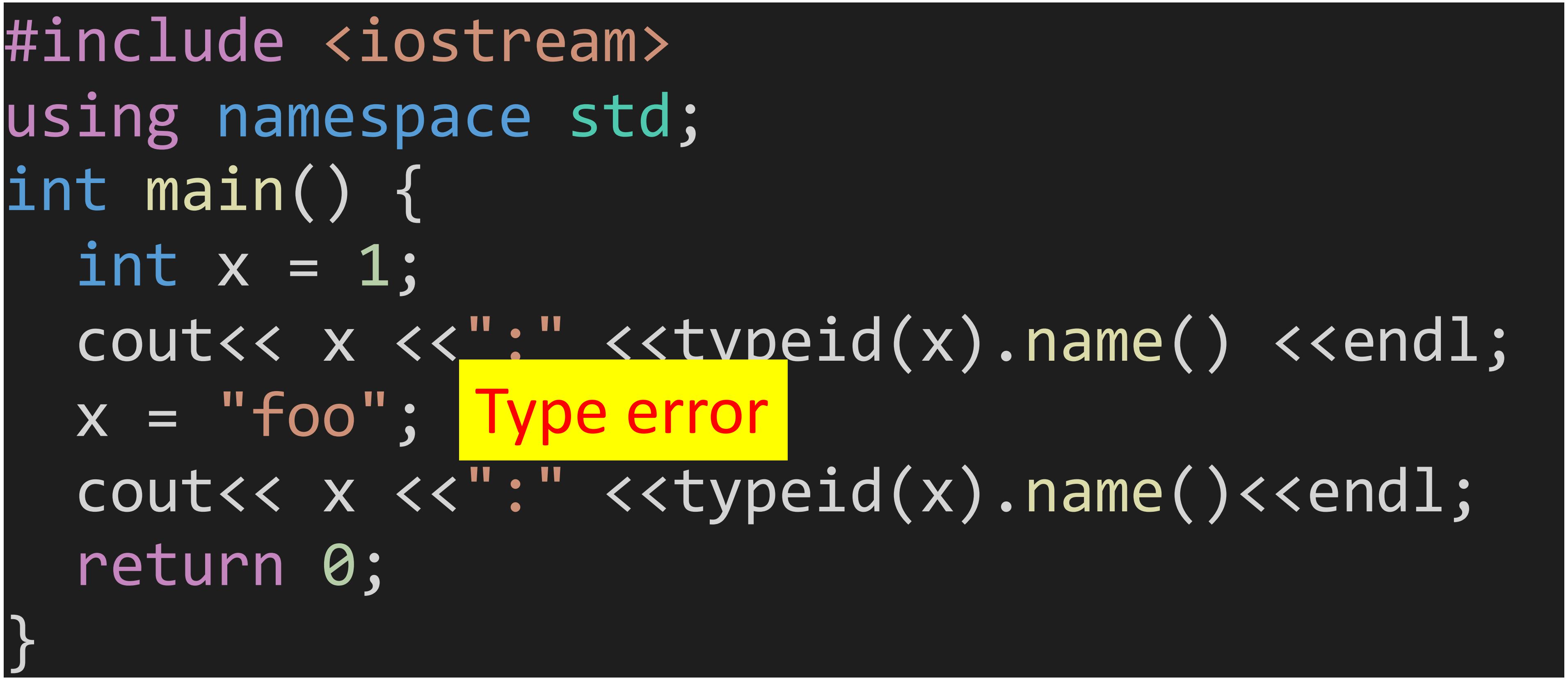
```
x = 1;
print("x value:",x,end=";");
print("x type:",type(x));
x = 1.0
print("x value:",x,end=";");
print("x type:",type(x));
```

x has different types during the execution

Output

```
x value: 1; x type: <class 'int'>
x value: 1.0; x type: <class 'float'>
```

C/C++: Static Typing



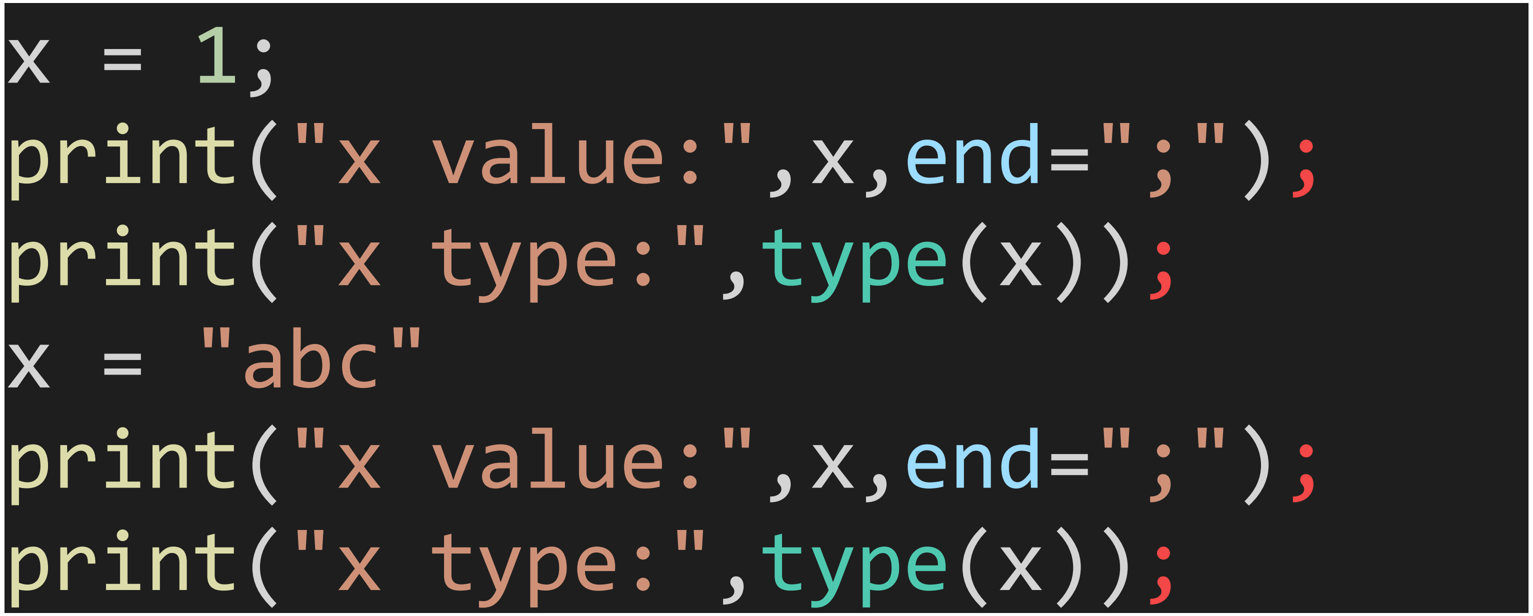
```
#include <iostream>
using namespace std;
int main() {
  int x = 1;
  cout<< x <<":" <<typeid(x).name() <<endl;
  x = "foo";
  cout<< x <<":" <<typeid(x).name()<<endl;
  return 0;
}
```

Type error

Output in VC++
19.00.23506 for x64

error C2440: '=': cannot convert from 'const char [4]' to 'int'

Python: Dynamic Typing



```
x = 1;
print("x value:",x,end=";");
print("x type:",type(x));
x = "abc"
print("x value:",x,end=";");
print("x type:",type(x));
```

x has different types during execution

Output

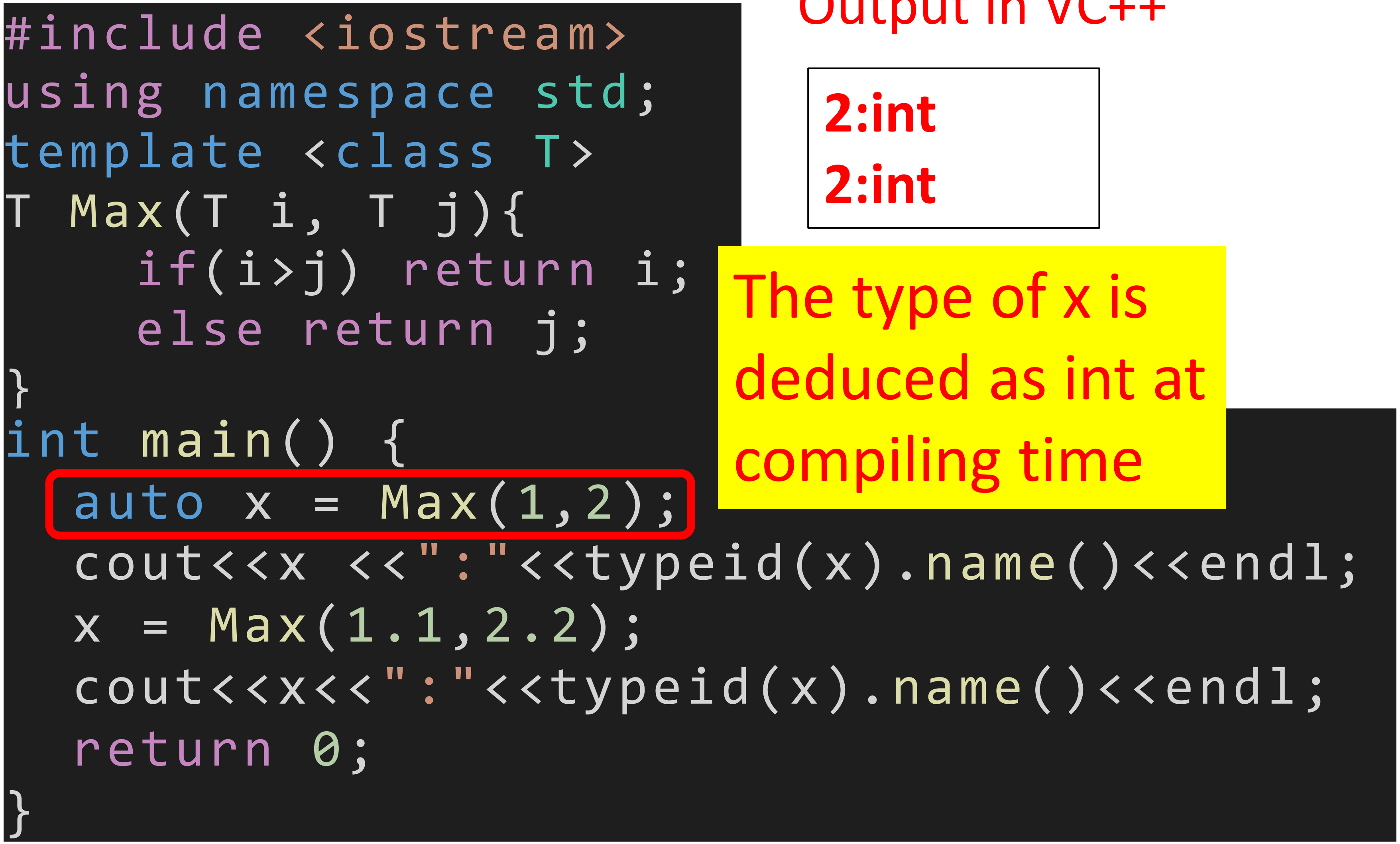
```
x value:1; x type:<class 'int'>
x value:abc; x type:<class 'str'>
```

“auto” specifier in C++11

- For variables, “auto” specifies that the type of the variable that is being declared will be automatically deduced from its initializer
- For functions, “auto” specifies that the return type is a trailing return type or will be deduced from its return statements (since C++14)
- For non-type template parameters, “auto” specifies that the type will be deduced from the argument (since C++17)

Still static typing and types are determined at compiling-time

“auto” specifier in C++11



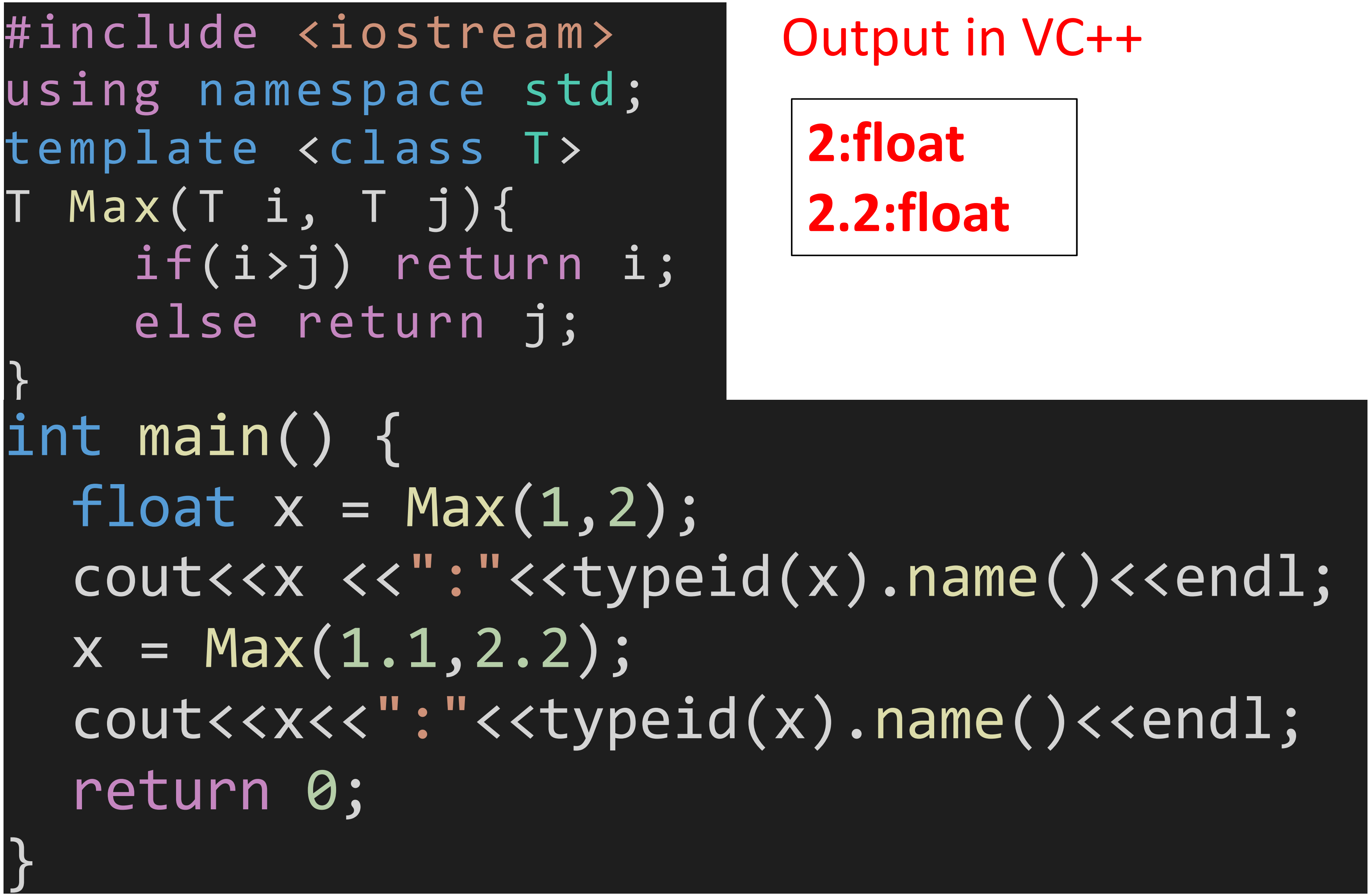
```
#include <iostream>
using namespace std;
template <class T>
T Max(T i, T j){
    if(i>j) return i;
    else return j;
}
int main() {
  auto x = Max(1,2);
  cout<<x <<":"<<typeid(x).name()<<endl;
  x = Max(1.1,2.2);
  cout<<x<<":"<<typeid(x).name()<<endl;
  return 0;
}
```

Output in VC++

```
2:int
2:int
```

The type of x is deduced as int at compiling time

“auto” specifier in C++11



```
#include <iostream>
using namespace std;
template <class T>
T Max(T i, T j){
    if(i>j) return i;
    else return j;
}
int main() {
  float x = Max(1,2);
  cout<<x <<":"<<typeid(x).name()<<endl;
  x = Max(1.1,2.2);
  cout<<x<<":"<<typeid(x).name()<<endl;
  return 0;
}
```

Output in VC++

```
2:float
2.2:float
```

C/C++ vs Python

- C/C++
  - Procedural + OO
  - Compilation
  - Bounded number
  - Weakly and statically typed language

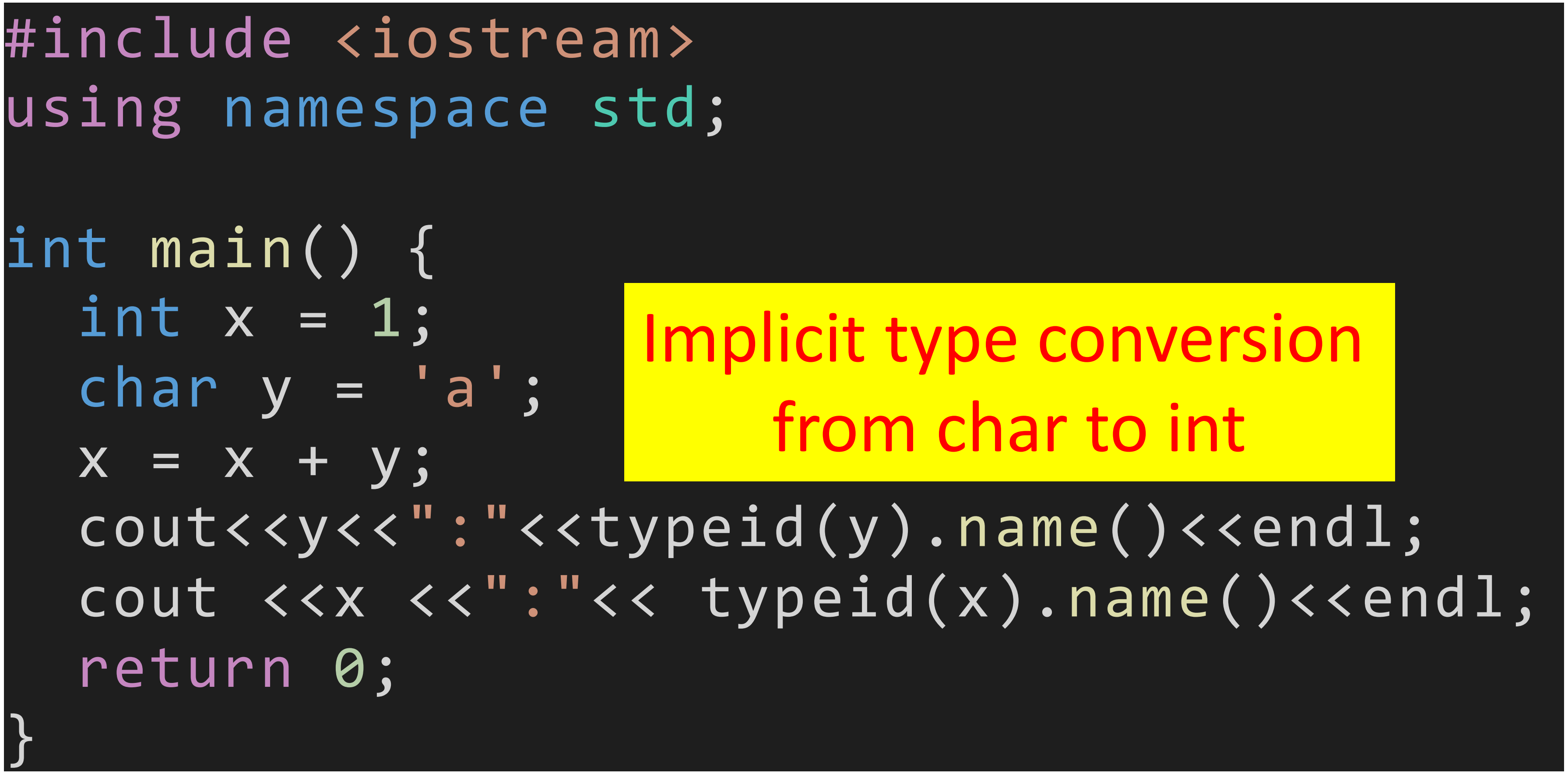
- Python
  - Procedural + OO (pure OO, every value is an object)
  - Interpretation
  - Unbounded number
  - Strongly and dynamically typed language

Strong Typing vs Weak Typing

- Strong Typing (Python)
  - A strongly typed language has stricter typing rules
  - Computations have to obey typing rules

- Weak Typing (C/C++)
  - A weakly typed language has looser typing rules
  - May produce unpredictable results or may perform implicit type conversion

C/C++: Weak Typing



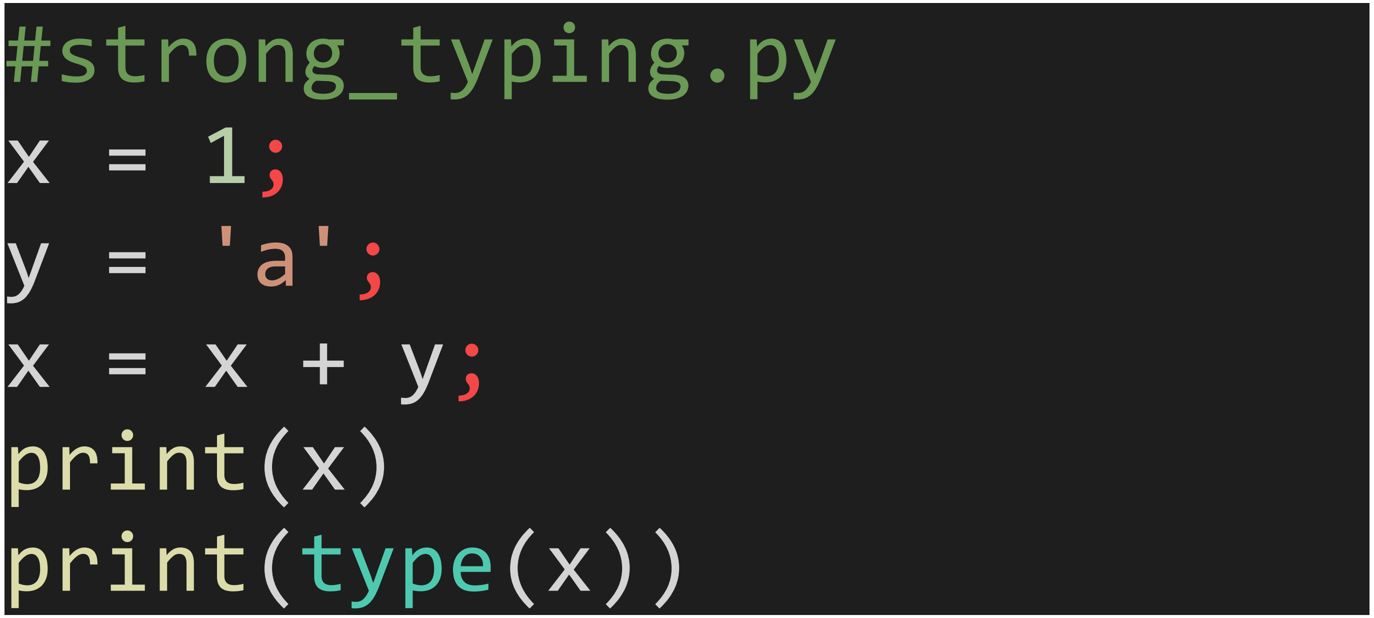
```cpp
#include <iostream>
using namespace std;

int main() {
   int x = 1;
   char y = 'a';
   x = x + y;
   cout<<y<<":"<<typeid(y).name()<<endl;
   cout <<x <<":"<< typeid(x).name()<<endl;
   return 0;
}
```

Implicit type conversion from char to int

Output in VC++

```
a:char
98:int
```

Python: Strong Typing



```
#strong_typing.py
x = 1;
y = 'a';
x = x + y;
print(x)
print(type(x))
```

```
TypeError: unsupported operand type(s)
for +: 'int' and 'str'
```

Recap

- Understand when use
  - C/C++
  - Python
- Simple I/O in Python
- Understand the difference between
  - C/C++
  - Python