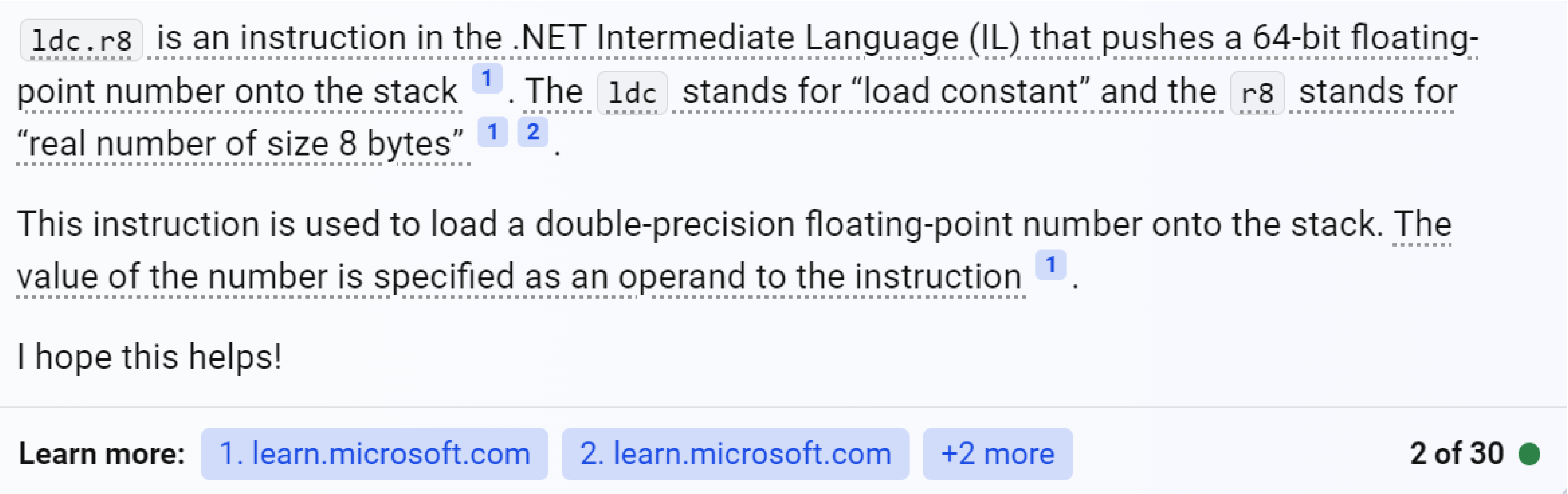
```csharp
namespace ConsoleApp1
{
    internal class Program
    {
        static void Main(string[] args)
        {
            double ab = 1.0;
            double y = 3 * (2 * ab + 55 + 3 * 6) * 8 / (2 * 3);
        }
    }
}
```

```
IL_0001: ldc.r8 1
IL_000a: stloc.0
IL_000b: ldc.r8 3
IL_0014: ldc.r8 2
IL_001d: ldloc.0
IL_001e: mul
IL_001f: ldc.r8 55
IL_0028: add
IL_0029: ldc.r8 18
IL_0032: add
IL_0033: mul
IL_0034: ldc.r8 8
IL_003d: mul
IL_003e: ldc.r8 6
IL_0047: div
IL_0048: stloc.1
```

https://sharplab.io/

`ldc.r8`



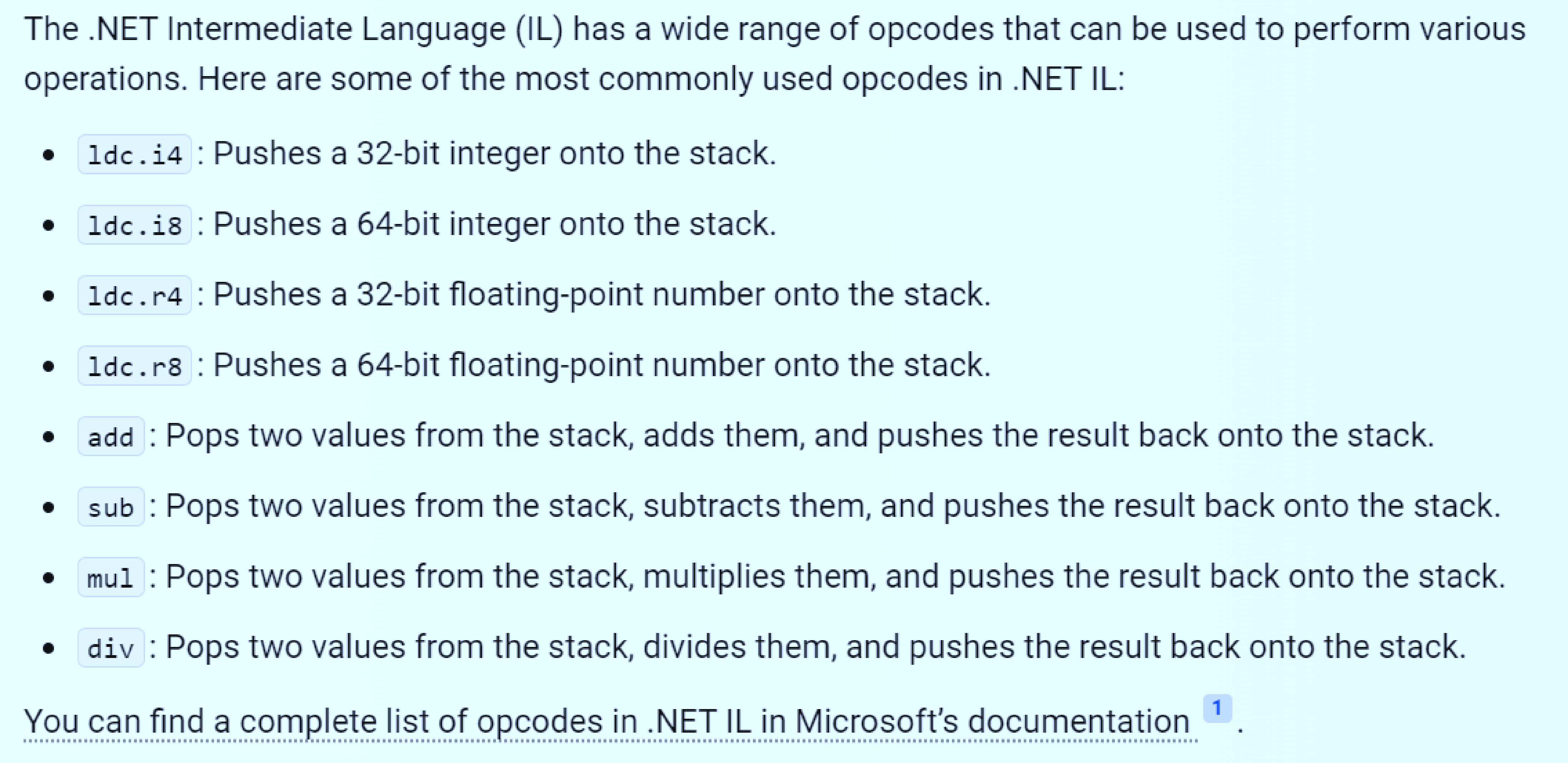
`ldc.r8` is an instruction in the .NET Intermediate Language (IL) that pushes a 64-bit floating-point number onto the stack [1]. The `ldc` stands for “load constant” and the `r8` stands for “real number of size 8 bytes” [1] [2].

This instruction is used to load a double-precision floating-point number onto the stack. The value of the number is specified as an operand to the instruction [1].

I hope this helps!

Learn more: 1. learn.microsoft.com 2. learn.microsoft.com +2 more

2 of 30 ●

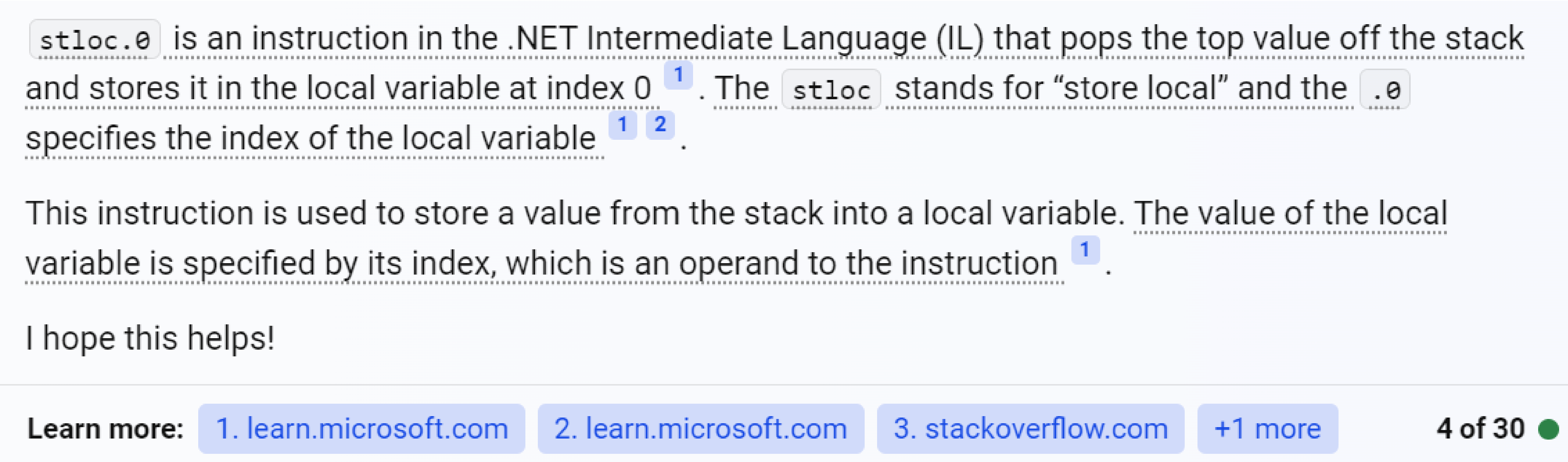


The .NET Intermediate Language (IL) has a wide range of opcodes that can be used to perform various operations. Here are some of the most commonly used opcodes in .NET IL:

- `ldc.i4` : Pushes a 32-bit integer onto the stack.
- `ldc.i8` : Pushes a 64-bit integer onto the stack.
- `ldc.r4` : Pushes a 32-bit floating-point number onto the stack.
- `ldc.r8` : Pushes a 64-bit floating-point number onto the stack.
- `add` : Pops two values from the stack, adds them, and pushes the result back onto the stack.
- `sub` : Pops two values from the stack, subtracts them, and pushes the result back onto the stack.
- `mul` : Pops two values from the stack, multiplies them, and pushes the result back onto the stack.
- `div` : Pops two values from the stack, divides them, and pushes the result back onto the stack.

You can find a complete list of opcodes in .NET IL in Microsoft's documentation [1].

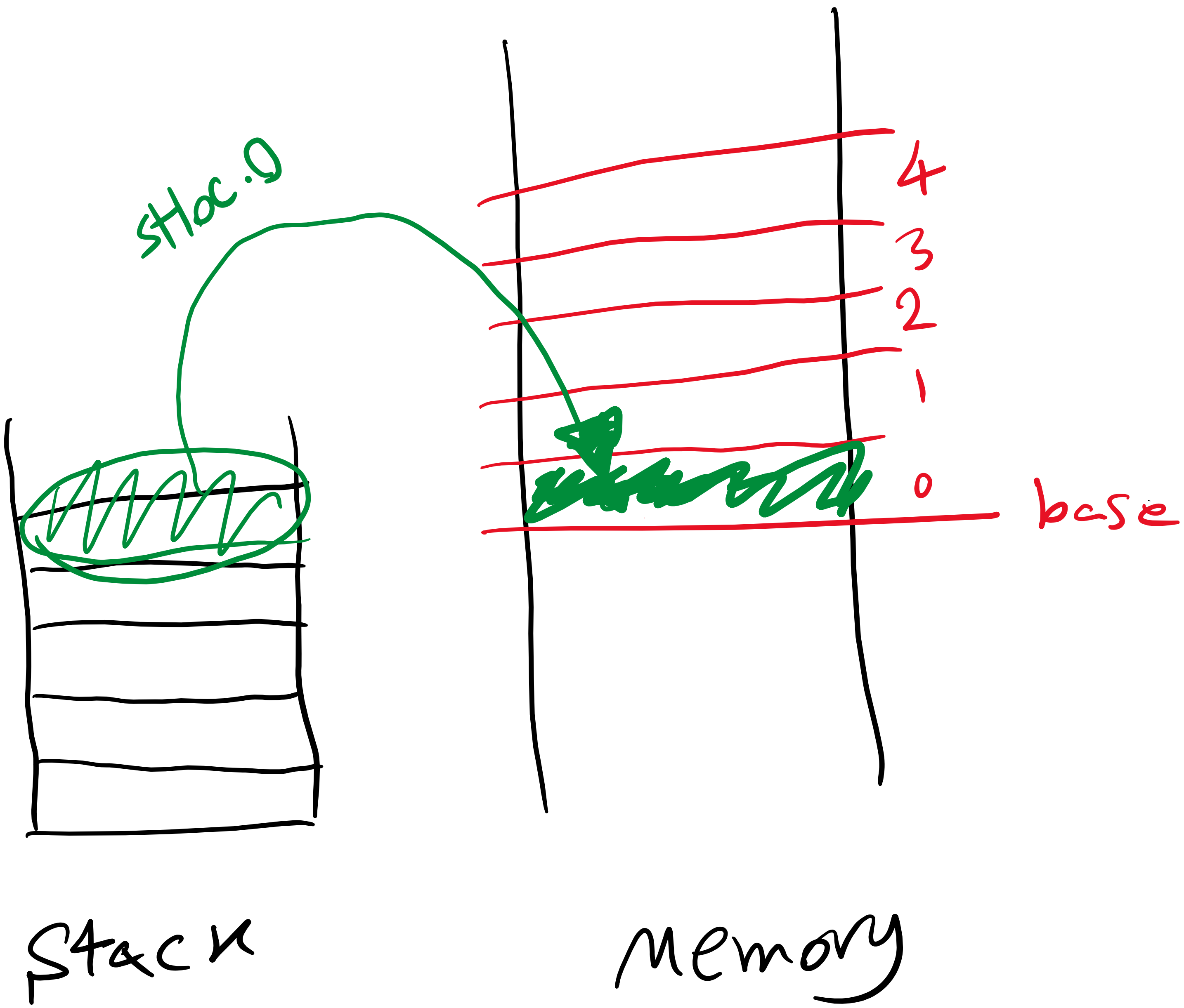
stloc.0



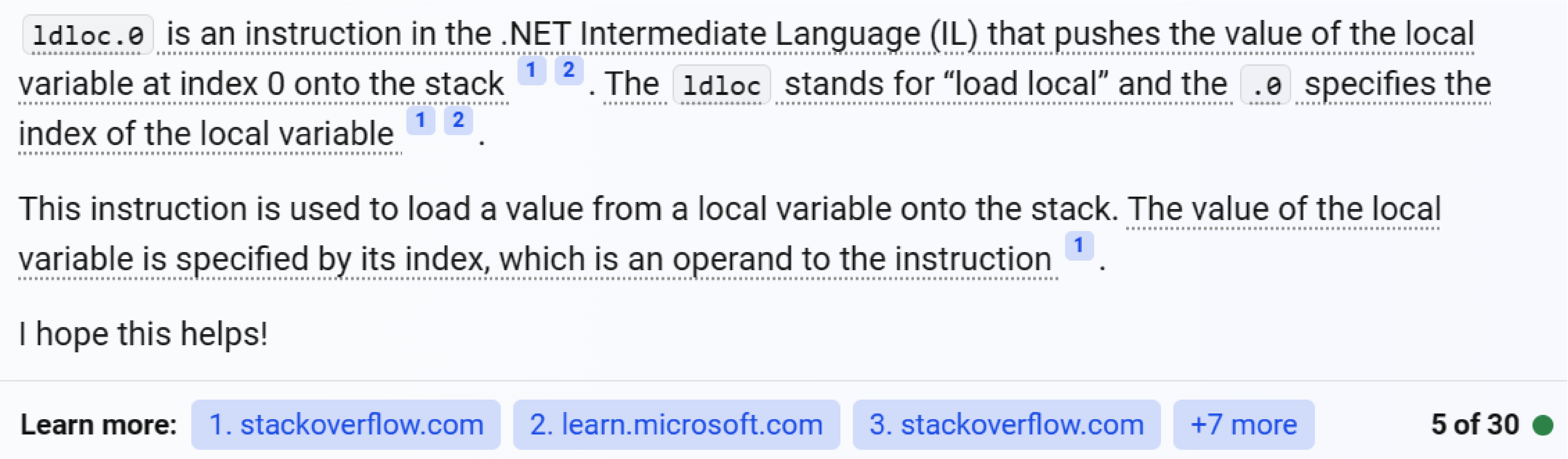
`stloc.0` is an instruction in the .NET Intermediate Language (IL) that pops the top value off the stack and stores it in the local variable at index 0 [1]. The `stloc` stands for “store local” and the `.0` specifies the index of the local variable [1] [2].

This instruction is used to store a value from the stack into a local variable. The value of the local variable is specified by its index, which is an operand to the instruction [1].

I hope this helps!


stloc.0
4
3
2
1
0
base
Stack
Memory

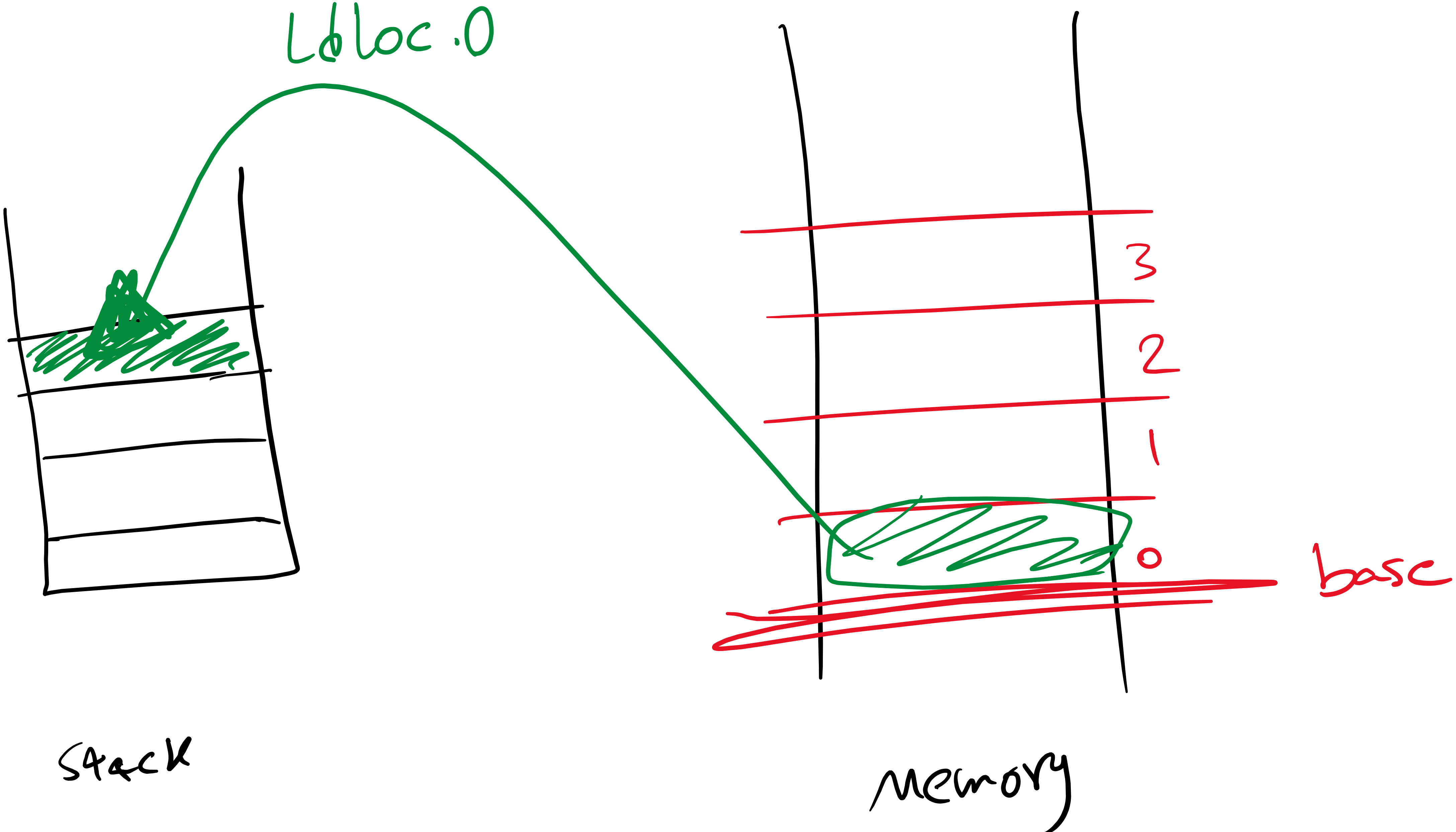
ldloc.0



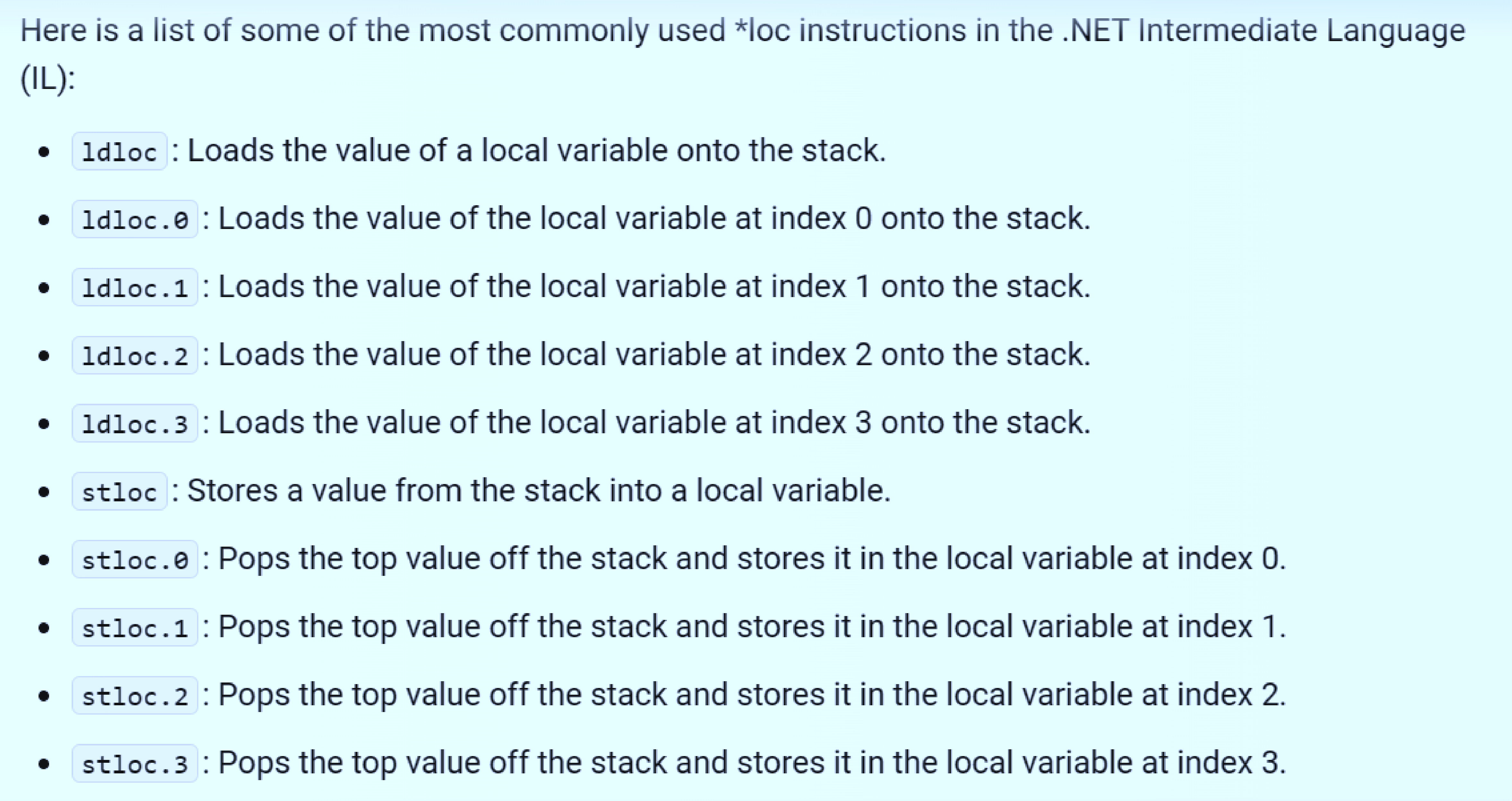
`ldloc.0` is an instruction in the .NET Intermediate Language (IL) that pushes the value of the local variable at index 0 onto the stack [1] [2]. The `ldloc` stands for "load local" and the `.0` specifies the index of the local variable [1] [2].

This instruction is used to load a value from a local variable onto the stack. The value of the local variable is specified by its index, which is an operand to the instruction [1].

I hope this helps!



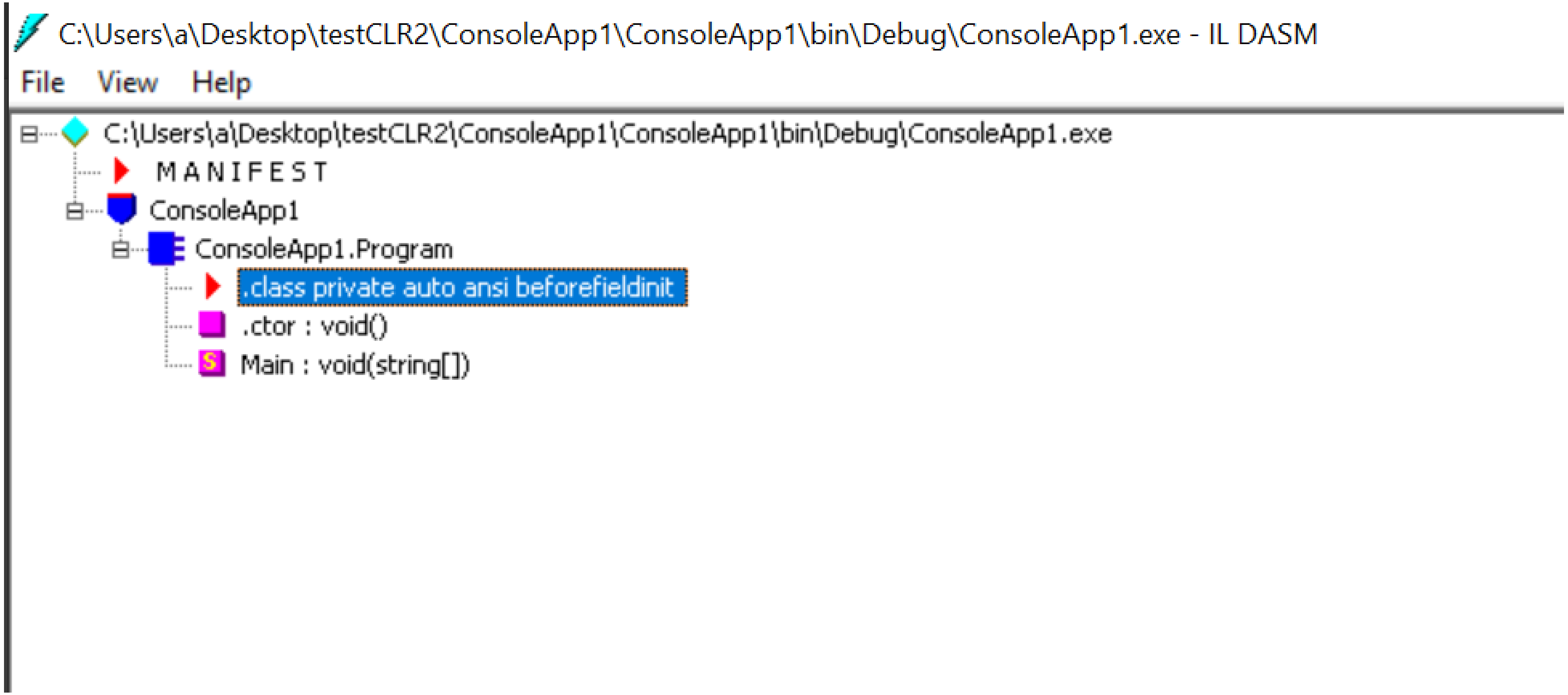
Ldloc.0
3
2
1
0
base
stack
memory



Here is a list of some of the most commonly used *loc instructions in the .NET Intermediate Language (IL):

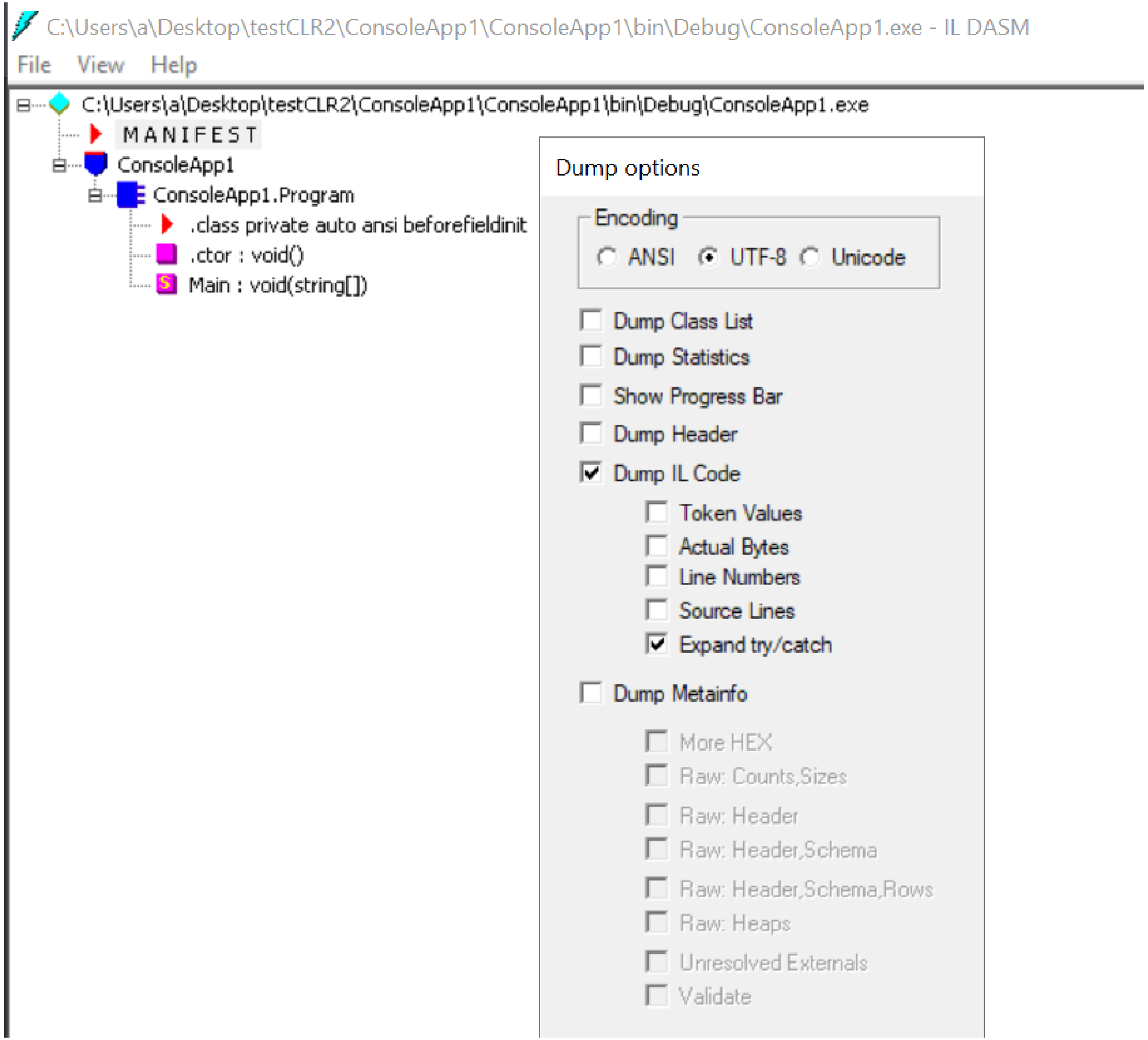
- `ldloc`: Loads the value of a local variable onto the stack.
- `ldloc.0`: Loads the value of the local variable at index 0 onto the stack.
- `ldloc.1`: Loads the value of the local variable at index 1 onto the stack.
- `ldloc.2`: Loads the value of the local variable at index 2 onto the stack.
- `ldloc.3`: Loads the value of the local variable at index 3 onto the stack.
- `stloc`: Stores a value from the stack into a local variable.
- `stloc.0`: Pops the top value off the stack and stores it in the local variable at index 0.
- `stloc.1`: Pops the top value off the stack and stores it in the local variable at index 1.
- `stloc.2`: Pops the top value off the stack and stores it in the local variable at index 2.
- `stloc.3`: Pops the top value off the stack and stores it in the local variable at index 3.

ILDASM.EXE



C:\Users\a\Desktop\testCLR2\ConsoleApp1\ConsoleApp1\bin\Debug\ConsoleApp1.exe - IL DASM
File
View
Help

```
C:\Users\a\Desktop\testCLR2\ConsoleApp1\ConsoleApp1\bin\Debug\ConsoleApp1.exe
    MANIFEST
    ConsoleApp1
        ConsoleApp1.Program
            .class private auto ansi beforefieldinit
            .ctor : void()
            Main : void(string[])
```



C:\Users\a\Desktop\testCLR2\ConsoleApp1\ConsoleApp1\bin\Debug\ConsoleApp1.exe - IL DASM

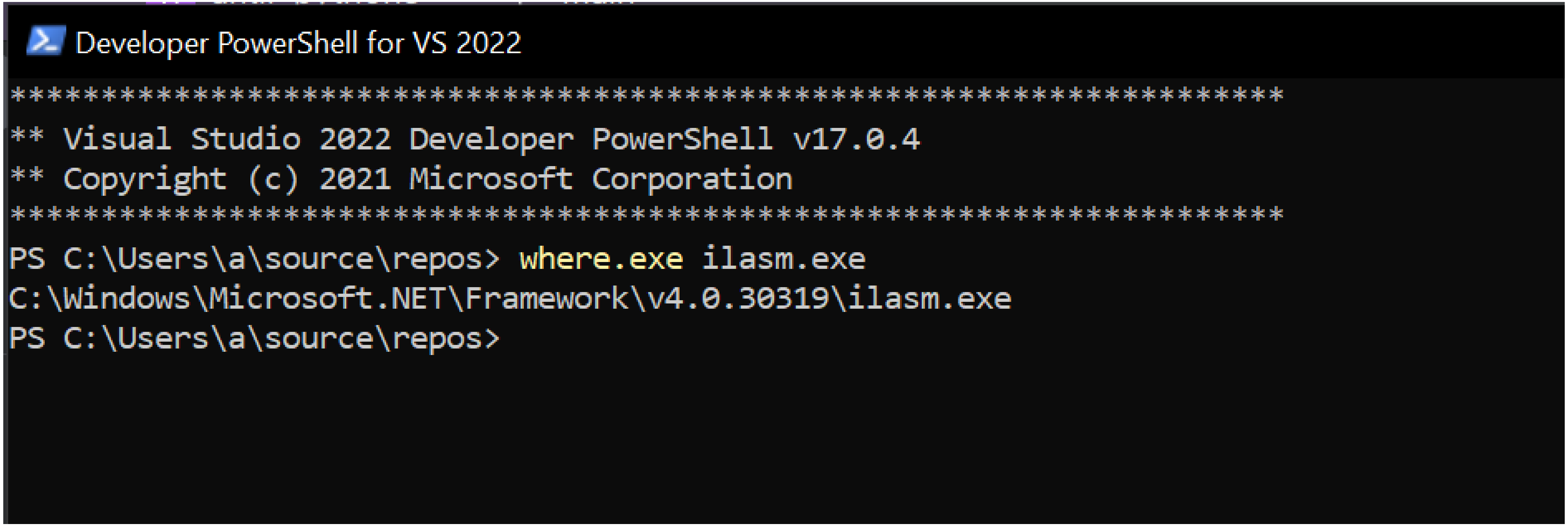
```
C:\Users\a\Desktop\testCLR2\ConsoleApp1\ConsoleApp1\bin\Debug\ConsoleApp1.exe
    MANIFEST
    ConsoleApp1
        ConsoleApp1.Program
            .class private auto ansi beforefieldinit
            .ctor : void()
            Main : void(string[])
```

Dump options

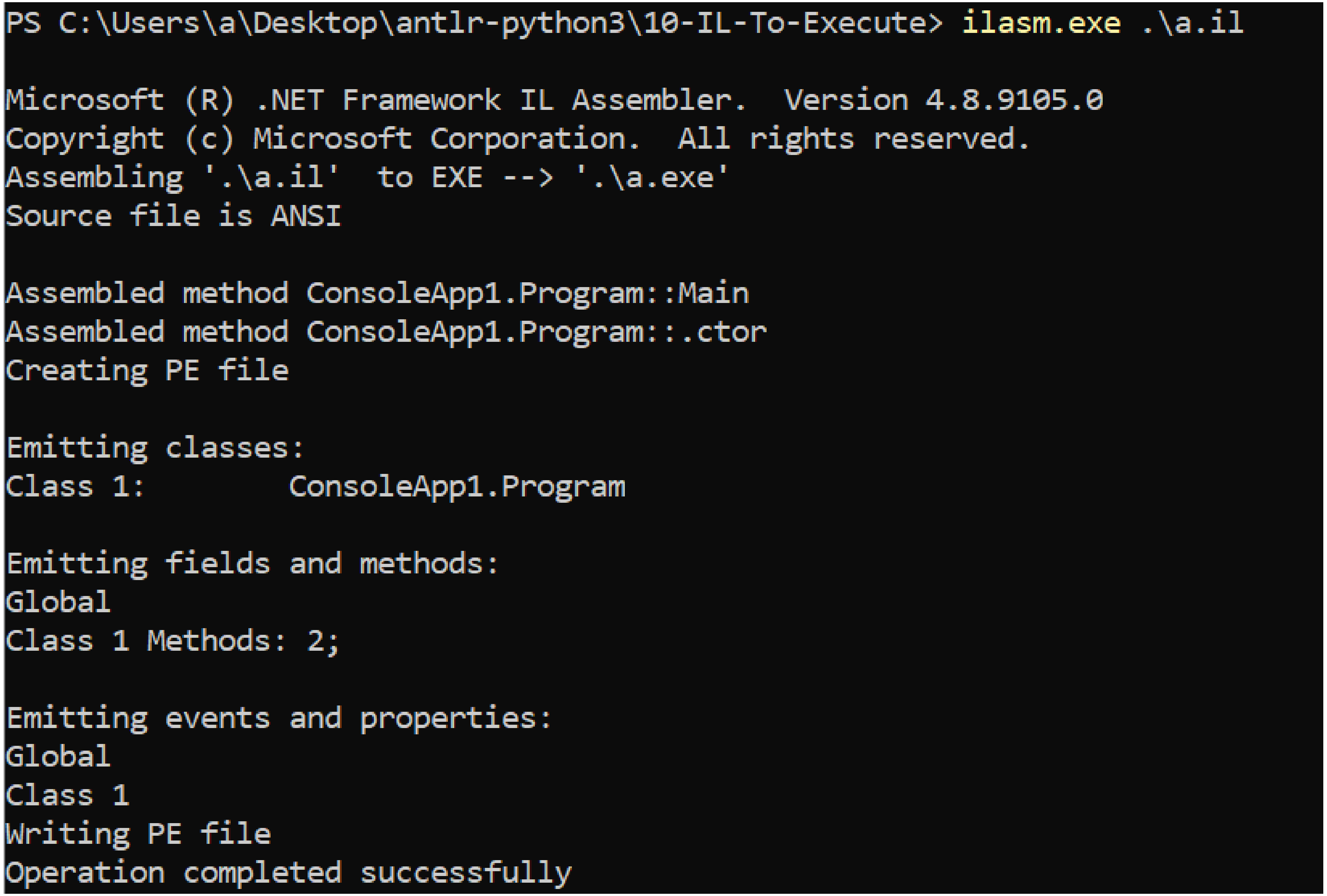
Encoding
○ ANSI ● UTF-8 ○ Unicode

- ☐ Dump Class List
- ☐ Dump Statistics
- ☐ Show Progress Bar
- ☐ Dump Header
- ☑ Dump IL Code
  - ☐ Token Values
  - ☐ Actual Bytes
  - ☐ Line Numbers
  - ☐ Source Lines
  - ☑ Expand try/catch
- ☐ Dump Metainfo
  - ☐ More HEX
  - ☐ Raw: Counts,Sizes
  - ☐ Raw: Header
  - ☐ Raw: Header,Schema
  - ☐ Raw: Header,Schema,Rows
  - ☐ Raw: Heaps
  - ☐ Unresolved Externals
  - ☐ Validate

ILASM.EXE



```
**********************************************************************
** Visual Studio 2022 Developer PowerShell v17.0.4
** Copyright (c) 2021 Microsoft Corporation
**********************************************************************
PS C:\Users\a\source\repos> where.exe ilasm.exe
C:\Windows\Microsoft.NET\Framework\v4.0.30319\ilasm.exe
PS C:\Users\a\source\repos>
```



```
PS C:\Users\a\Desktop\antlr-python3\10-IL-To-Execute> ilasm.exe .\a.il

Microsoft (R) .NET Framework IL Assembler.  Version 4.8.9105.0
Copyright (c) Microsoft Corporation.  All rights reserved.
Assembling '.\a.il'  to EXE --> '.\a.exe'
Source file is ANSI

Assembled method ConsoleApp1.Program::Main
Assembled method ConsoleApp1.Program::.ctor
Creating PE file

Emitting classes:
Class 1:        ConsoleApp1.Program

Emitting fields and methods:
Global
Class 1 Methods: 2;

Emitting events and properties:
Global
Class 1
Writing PE file
Operation completed successfully
```

```
> ilasm.exe .\a.il
> .\a.exe
```