10.2D Other Language Comparison

Chris Dilger May 8, 2018

1 History of Python and C#

In understanding the differences between these languages first investigating their historical roots, both the needs they fulfilled and the guiding philosophies of their respective creators heavily influenced the design of each of these languages.

Python, created by Guido Van Rossum in the late 1980's was intended to be a language that was easy to use, highly extensible and include a substantial set of inbuilt functionality. To meet these criterion, we have an interpreted, dynamically typed language which allows programmers to develop using an Object Oriented Programming approach. Also providing features to support the OOP paradigm, C# is a statically typed JIT compiled language with a strong historical focus on the Windows platform. Progressively Library code included in the .NET framework centres around this Windows centred design focus when providing existing objects, methods and architectures for programmers to build from. Each language from an OOP perspective achieves largely the same goals whilst implementing the principles of OO programming using alternative methods.

2 Dynamic and Static Typing Systems

Type checking is a language feature that ensures data stored in memory is only accessed or manipluated in such a way that makes sense for that type of data. Preventing an 'average()' method from accepting a string prevents unintended behaviour.

Python delays type checking until runtime, or in the Fig. 1 final step during runtime. This means that in practice, if we passed Square("not a number") the instance variable .length will be of type string. The program will not be able to tell us there is a problem until runtime, when an error will be thrown. This has other side effects when object oriented concepts are considered. Polymorphism for instance is largely left up to the programmer, as the language does not provide much support in terms of explicit syntax.

Duck typing comes from the phrase: "If it looks like a duck and quacks like a duck, it's a duck" (*programming languages - What is duck typing?* 2017). This means that, if we suppose that anything with a .Area() is a shape, then by this definition we can handle a Square() and a Circle() as shapes provided they define this method. There is no formal interface definition like there is in C# which allows greater flexibility

Listing 1: Python Class Definition - Dynamic Typing Example

```
class Square:
```

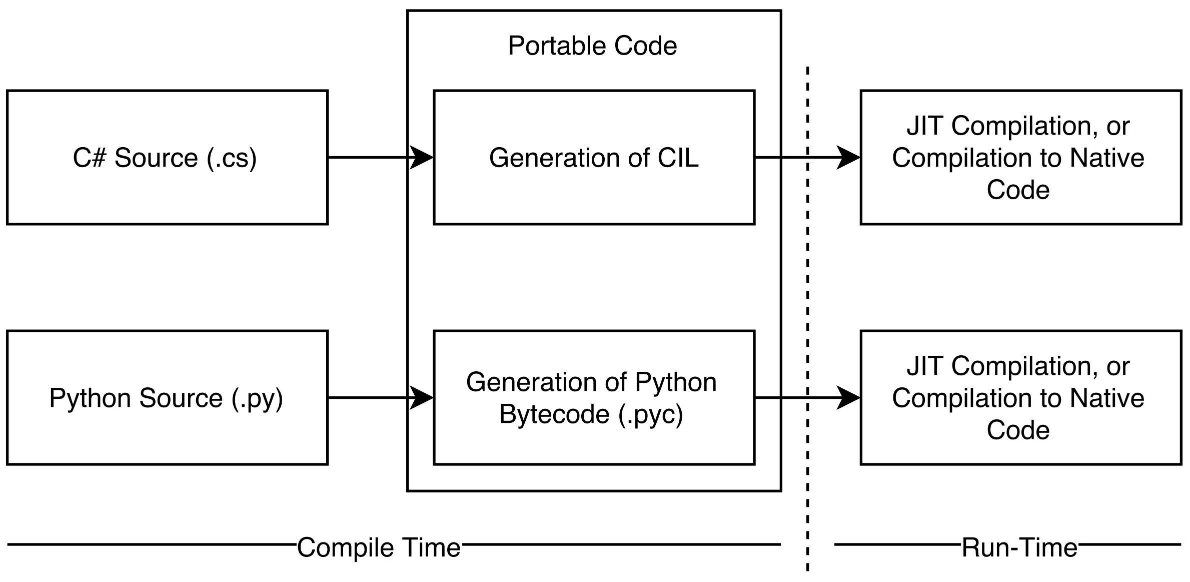
Portable Code
C# Source (.cs)
Generation of CIL
JIT Compilation, or Compilation to Native Code
Python Source (.py)
Generation of Python Bytecode (.pyc)
JIT Compilation, or Compilation to Native Code
Compile Time
Run-Time

Figure 1: Python and C# Program Execution Steps

```
    def __init__(self, length):
        self.length = length

    def area(self):
        return self.length ** 2

    def __str__(self):
        return "Area: " + str(self.area())
```

In contrast, we see the statically typed C# language depending on language constructs to attempt to catch type mismatch errors much earlier in the development cycle. Whilst this is significantly more verbose as shown in Listing 2 this syntax ensures that compilation will fail. (*roslyn* 2017). As seen in the constructor, the declaration that this is a Square means that if we tried to create a collection of type Shape, we would first have to define an abstract class or interface, have each class implement that interface via some C# syntax and then do the same on another Shape, perhaps a Circle. In C# the same effect will require at least one more file, and additional syntax on each of the derived classes.

Listing 2: C# Code

```
namespace CSharpClasses
{
    public class Square
    {
        public int Length { get; set; }
        public Square(int length)
        {
            Length = length;
        }

        public int Area()
        {
            return Length ^ 2;
        }

        public override string ToString()
        {
            return $"Area: {this.Area()}";
        }
    }
}
```

3 Interpreted and Compiled Conceptually

Interpreted and compiled are descriptions of a language that are not tied to the language specification itself, but to the implementation of that language. (Power and Rubinsteyn 2013; *csharplang* 2017; Mono 2017) Whilst at first the definition of a compile language seems to separate compiled lanugages from interpreted languages, this distinction becomes rather unclear relatively quickly. Consider compilation as the process of converting some higher level language into machine level instructions, before runtime. In contrast, an interpreted language largely relies on the conversion between program code and machine code at runtime. Thus typically performance of compiled languages are better than interpreted. Speed is in this case at the expense of code portability, as machine-code is typically less portable.

3.1 Interpreted Languages: Python

As the endpoint for all code is the CPU which will ultimately perform code execution, the line between interpreted and compiled languages blur as the code execution process is analysed in more depth. Consider Listing 3, which corresponds to step 2 in 1. This step may occur at runtime if no .pyc file is generated beforehand. Here we see the Python 3 bytecode in human readable format, corresponding to the Square object in 1.

Instructions in this bytecode (LOAD_FAST, STORE_ATTR, RETURN_VALUE) are then interpreted by an implementation of the Python Virtual Machine, which in the CPython implementation translates these instructions to lower level C instructions (Meijer and Drayton n.d.).

Listing 3: Python Compiled Bytecode (.pyc) interpreted by the PVM (ex. CPython)

```
Disassembly of __init__:
  4           0 LOAD_FAST                1 (length)
              3 LOAD_FAST                0 (self)
              6 STORE_ATTR               0 (length)
              9 LOAD_CONST               0 (None)
             12 RETURN_VALUE

Disassembly of __str__:
 10           0 LOAD_CONST               1 ('Area: ')
              3 LOAD_GLOBAL              0 (str)
              6 LOAD_FAST                0 (self)
              9 LOAD_ATTR                1 (area)
             12 CALL_FUNCTION            0 (0 positional, 0 keyword pair)
             15 CALL_FUNCTION            1 (1 positional, 0 keyword pair)
             18 BINARY_ADD
             19 RETURN_VALUE

Disassembly of area:
  7           0 LOAD_FAST                0 (self)
              3 LOAD_ATTR                0 (length)
              6 LOAD_CONST               1 (2)
              9 BINARY_POWER
             10 RETURN_VALUE
```

3.2 JIT Compiled Language: C#

C# is compiled by Roslyn into Microsoft's Common Intermediate Language (IL) to be interpreted by the .NET runtime. In 4 we see a sample disassembled RE (Runtime Executable). IL is a low level Object Oriented programming language which is targeted by a C#, VB and a collection of other languages. .NET runtime is a Virtual Machine, which will Just In Time (JIT) compile IL into Native Machine Code to be executed by the CPU. Several striking similarities exist between the JIT compilation of the .NET runtime and the interpretation of the PVM.

Since there are a multitude of languages including F#, Visual Basic, Visual C++, Perl and COBOL that can all target the IL, we will focus specifically on the conversion of C# code into IL. From Microsoft documentation, while the .NET runtime supports a range of language features it's ultimately the compiler and not the runtime that allows you to use these features (rpetrusha 2017[a]). Thus the implementation of OOP principles, while IL is a low level Object Oriented language it is the conversion from C# into IL done by the Roslyn compiler that largely determines the features of the language.

Listing 4: CIL Intermediate Language

```
.namespace CSharpClasses
{
  .class private auto ansi beforefieldinit CSharpClasses.Program
    extends [mscorlib]System.Object
  {
    // Methods
    .method private hidebysig static
      void Main (
        string[] args
      ) cil managed
    {
      // Method begins at RVA 0x2050
      // Code size 16 (0x10)
      .maxstack 1
      .entrypoint
      .locals init (
        [0] class CSharpClasses.Square myShape
      )

      IL_0000: nop
      IL_0001: ldc.i4.5
      IL_0002: newobj instance void CSharpClasses.Square::.ctor(int32)
      IL_0007: stloc.0
      IL_0008: ldloc.0
      IL_0009: call void [mscorlib]System.Console::WriteLine(object)
      IL_000e: nop
      IL_000f: ret
    } // end of method Program::Main
//Ouptut truncated
```

4 Practical Differences In the Application of OOP Concepts

To the programmer the difference between C# and Python in terms of how Object Oriented Design Principles are executed are still significant despite apparent similarities. Aside from syntactical differences when creating objects using dynamic types lends itself to faster development and less rework in the early stages of development. As we see in Listing 1 the self object is assigned new properties. This makes some patterns easier to implement.

For example the factory pattern is relatively trivial in Python - simply return a new object from a function, or even dynamically create a new object with the required method. In C# similar functionality exists though objects must be explicitly created. From a programmer's perspective, when implementing OOP Design Patterns the rapid and easy appeal of Python's more permissive and less verbose syntax allow quicker conceptual testing at the expense of catching errors at compile time.

4.1 Access Levels

C# supports Access levels, which place restrictions on which code can access certain objects, their methods and their properties. For example, a public method can be accessed by any other object, whereas a method declared private can only be accessed by code implementing the object itself. Similarly a protected access level prevents public access, but can be accessed by derived classes. Without any formal form of inheritance or explicit interfaces Python has no such restrictions. Python does not enforce access levels at all, every property, method and field on an object is public. Instead, a convention has been adopted involving an "_" which signals to developers the code is only intended to be run by the object itself. Python leaves the responsibility of not violating the principle of encapsulation to the developer, where C# would require the access level to be explicitly changed before allowing unintended access.

4.2 Unit Testing and Mocking

Validating the code produced is almost as important as writing the code itself. If the functionality of the code in question has not been tested, how do we know how it will behave in a real program? Unit tests are designed to explicitly specify the functionality of code, however testing isolated parts of code are often impossible without some other objects, especially in cases where an object composition relationship exists. In this case, Python allows dynamic objects to be created to 'mock' the functionality of objects the tested object uses. C# however relies on extensive Mocking libraries, or verbose syntax or in the worst case full implementation versions of the objects themselves.

5 Conclusion

C# and Python have a range of features that allow them to implement OOP Design Principles. C# implements a variety of control measures to encourage good OOP design, where Python generally releases restrictions and code verbosity at the expense of allowing poorer design decisions. C# targeting a corporate user base which are more compatible with extensive management of risk and are more willing to accept a cost to reduce risk where Python targets a community which favors a more open approach to implementing OOP Design Principles. From the analysis of these two languages, the existence of cases whereby one language should have some advantage over the other when it comes to implementation of OOP Principles based on one of a number of properties that have been outlined. More than being about the conversion of code to native machine code that can execute instructions, programming languages are about how programmers can express solutions to software engineering problems.

Cases exist where each language may have an advantage over the other, which is only one of many design decisions that must be taken into account when solving any problem with an OOP approach.

References

Allison Kaptur (2017). *500 Lines or Less | A Python Interpreter Written in Python*. URL: `http://www.aosabook.org/en/500L/a-python-interpreter-written-in-python.html` (visited on 10/21/2017).

BillWagner (2017). *Lambda Expressions (C# Programming Guide)*. URL: `https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/statements-expressions-operators/lambda-expressions` (visited on 10/19/2017).

Brian R. Bondy (2017). *python - What are "first class" objects? - Stack Overflow*. URL: `https://stackoverflow.com/questions/245192/what-are-first-class-objects` (visited on 10/18/2017).

Clark, Dan (2013). *Beginning C# Object-Oriented Programming*. Google-Books-ID: lEKnfHgJ8ewC. Apress. 372 pp. ISBN: 978-1-4302-4936-8.

csharplang (2017). *csharplang: The official repo for the design of the C# programming language*. original-date: 2016-12-09T01:45:27Z. URL: `https://github.com/dotnet/csharplang`.

erikdietrich (2017). *The history of C# - C# Guide*. URL: `https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-version-history` (visited on 10/22/2017).

General Python FAQ — Python 3.6.3 documentation (2017). URL: `https://docs.python.org/3/faq/general.html` (visited on 10/18/2017).

JIAN HUANG (2017). *A Brief History of Object-Oriented Programming*. URL: `http://web.eecs.utk.edu/~huangj/CS302S04/notes/oo-intro.html` (visited on 09/20/2017).

Mahesh Alle (2017). *Code Execution Process*. URL: `http://www.c-sharpcorner.com/uploadfile/8911c4/code-execution-process/` (visited on 10/21/2017).

Meijer, Erik and Peter Drayton (n.d.). “Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages”. In:

Mono (2017). *C# REPL | Mono*. URL: `http://www.mono-project.com/docs/tools+libraries/tools/repl/` (visited on 10/22/2017).

.NET Framework Class Library () (2017). URL: `https://msdn.microsoft.com/en-us/library/gg145045(v=vs.110).aspx` (visited on 10/19/2017).

Power, Russell and Alex Rubinsteyn (2013). “How fast can we make interpreted Python?” In: *arXiv:1306.6047 [cs]*. arXiv: `1306.6047`. URL: `http://arxiv.org/abs/1306.6047`.

programming languages - What is duck typing? (2017). *programming languages - What is duck typing? - Stack Overflow*. URL: `https://stackoverflow.com/questions/4205130/what-is-duck-typing` (visited on 10/22/2017).

roslyn (2017). *roslyn: The .NET Compiler Platform ("Roslyn") provides open-source C# and Visual Basic compilers with rich code analysis APIs*. original-date: 2015-01-11T02:39:03Z. URL: `https://github.com/dotnet/roslyn`.

rpetrusha (2017[a]). *Common Language Runtime (CLR)*. URL: `https://docs.microsoft.com/en-us/dotnet/standard/clr` (visited on 10/21/2017).

— (2017[b]). *Managed Execution Process*. URL: `https://docs.microsoft.com/en-us/dotnet/standard/managed-execution-process` (visited on 10/23/2017).

Smith, J. E. and Ravi Nair (2005). “The architecture of virtual machines”. In: *Computer* 38.5, pp. 32–38. ISSN: 0018-9162. DOI: `10.1109/MC.2005.173`.