**Ruby**

Yukihiro “Matz” Matsumoto released the first version of the Ruby programming language in 1995 in Japan. Ruby is dynamic, reflective, object-oriented, and multi-paradigm. Let’s examine those terms.

**Dynamic**

A dynamic programming language is one that can execute many common programming behaviors at runtime (when the interpreter executes the program) that static programming languages perform during compilation, when a program--the compiler--translates high-level source code to a lower-level language (e.g., assembly language or machine code). Dynamic programming languages can compile code at runtime and execute code during compile-time; the compilation and runtime phases are muddled. Though this flexibility enables metaprogramming, more abstraction, and less code, it comes at the expense of execution speed.

Ruby, like most dynamic programming languages, is also dynamically typed. In dynamically typed languages, the language enforces type constraints at runtime; whereas, in statically typed languages, the language enforces type constraints at compile-time. The following method invocation in Ruby runs successfully at runtime:

def erroneous\_addition(x)

if x > 99999

print "Luckily I'm a valid statement!"

else

print "The Ruby interpreter is about to be a complainy-pants!" + x

end

end

erroneous\_addition(Float::INFINITY)

Ruby doesn't check the validity of erroneous\_addition until runtime. The following invocation causes an error because the interpreter encounters the erroneous branch of the conditional statement:

erroneous\_addition(1) # TypeError: no implicit conversion of Integer into String

Merely defining an equivalently erroneous method in a statically typed language like Java will result in a compilation error, namely incompatible types: String cannot be converted to int.

class StaticallyTypedError {

// Many statically typed programming languages require the programmer to specify variable types.

// The first int refers to the type of the return value of the method.

// The second refers to the type of the x parameter.

public static int erroneousAddition(int x) {

if (x > 99999)

System.out.println("Luckily I'm a valid statement!");

else

System.out.println("Java is already angry!" + x);

return "Java won't even let this method slide uninvoked!";

}

};

**Reflective**

Reflection is common among dynamic programming languages. Unlike mere type introspection, which refers to the ability of a program to examine the type and state of an object at runtime, reflection refers to ability of a program to manipulate its own structure and behavior as data. In object-oriented reflective languages like Ruby, reflection permits not only the examination of classes, modules, and methods, but also instantiation and method invocation. Reflection makes possible much of metaprogramming. Here's an example adapted from Wikipedia of non-reflective and reflective call sequences in Ruby:

# nonreflective

obj = Foo.new

obj.hello

# reflective

class\_name = "Foo"

method\_name = :hello

# The program can inspect and modify source code constructions such as classes

# and methods at runtime. It can even convert strings or symbols matching

# symbolic names of those constructions into references to them.

obj = Object.const\_get(class\_name).new

obj.send method\_name

# The program can evaluate a string as if it were source code at runtime.

eval "Foo.new.hello"

**Object-Oriented**

Object-oriented programming is a programming paradigm that privileges objects rather than actions and data rather than functions or logic. Adherents to OOP conceive of a program as a society of objects that receive messages that they then use to perform their own discrete operations. Objects typically contain data in fields known as attributes and a set of associated methods that may access and manipulate these attributes.

The most popular object-oriented languages, such as Java, Perl, Python, Ruby, Objective-C, Smalltalk and Swift are class-based. Objects are instances of classes, which usually determine type. Classes define the data format and methods available to their instances or to themselves.

Classes often inherit from other classes in an "is-a-type-of" relationship. The Dog class (the child class) might inherit from the Pet class (the parent class). The child class inherits data definitions and methods from the parent class, allowing for recycling. The child class may overwrite these definitions and methods as well as add its own, thereby customizing and extending its parent.

Encapsulation is a principle of object-oriented design. It dictates that the programmer:

1. Place all code concerned with a particular set of data in one class.
2. Hide methods and data essential only to a class's internal workings (this technique is called information hiding). Conversely, the programmer should expose via methods only what's necessary to a class's relationships with other classes, thereby preventing external interference and misuse.

**Multi-Paradigm**

Although Ruby is perhaps the object-oriented programming language par excellence, it also supports procedural and functional programming.

Procedural programming privileges procedure invocations, i.e., method calls. In procedural programming, the programmer specifies a series of procedures that operate on data structures in systematic order. Object-oriented programming joins data structures and methods into "objects" that operate on themselves. Defining and invoking methods outside of a class in Ruby approximates procedural programming because these methods are part of a single root object, limiting object-oriented design.

Functional programming privileges pure functions, those whose return value is only determined by the input, without side effects such as changes in state. Adherents of functional programming conceive of computation as the evaluation of mathematical functions. Via blocks, Ruby shares many features of functional programming languages. Ruby features anonymous functions, lexical closures (functions that capture variables in the context where they're defined), and higher-order functions (functions that can accept functions as arguments and/or return functions). As of Ruby 2.0, Ruby even supports lazy enumeration!

**Philosophy**

In 2008, Matz declared that the primary purpose of Ruby is to "help every programmer in the world to be productive, and to enjoy programming, and to be happy."

Ruby follows the Principle of Least Astonishment: it attempts to minimize confusion for experienced users. Much of Ruby reads like English. It's designed to be expressive, powerful, elegant, and flexible. There are myriad ways to do almost anything in Ruby, a design decision inherited from Perl.

**Ruby vs. Python**

Python is perhaps the most similar language to Ruby. Both are dynamic, reflective, object-oriented, multi-paradigm scripting languages. They even use similar syntax! Here are the primary linguistic differences:

1. Python doesn't support blocks.
2. Python has a richer set of data structures.
3. Python is inflexible: there's one best way of doing things!
4. Whitespace is syntactically significant in Python.
5. Python is less object-oriented. It has primitive data types, which are more fundamental than objects (not everything in Python is an object!), and many of its object-oriented features were late additions.
6. Ruby values elegance and "magic" over clarity. Here's how to determine the time one month from now in Ruby versus Python:
7. # RUBY
8. require 'active\_support/all'

1.month.from\_now

# PYTHON

from datetime import datetime

from dateutil.relativedelta import relativedelta

new\_time = datetime.now() + relativedelta(months=1)

The most significant difference between Ruby and Python is their communities. For better or worse, Ruby's popularity is tied to Rails. Hence Rails is the testing ground for many innovations in web development. Airbnb, Apple, Github, Groupon, Shopify, and Twitter all use Rails.

Python developers tend to be more conservative in general. Rubyists enjoy many new linguistic features before Python developers catch on.

Because of the popularity of Rails, the fact that Python is older, and the early porting of statistical libraries in R to Python, Python has a more diverse community, including many more proponents in data science and academia. Ruby has no equivalent to Python's SciPy, NumPy, Matplot lib, or NLTK (Natural Language Toolkit) libraries.

Here's an excellent infographic from UpGuard comparing the two languages: [Python versus Ruby](https://assets.aaonline.io/fullstack/ruby/assets/python_vs_ruby.jpg)

**Vocabulary**

**Test-Driven Development (TDD)**

TDD is a process for writing software. Its basic steps are as follows:

1. Add a test
2. Run all the tests and make sure the new test fails
3. Write code
4. Run all the tests and make sure the new test passes
5. Refactor the code.

**Behavior-Driven Development (BDD)**

BDD is an extension of TDD. It structures the testing process through the use of user stories - a description of the user's use of the feature being developed. The goal of the user story is to help focus the developer on what to test and what not to test.

**Unit Test**

Unit testing is a way of testing an application by breaking the application down into its smallest constituent parts and testing each part in isolation. In order to test parts of an application that rely on other code, unit tests typically use mock objects to make sure that problems elsewhere in the code won't cause the tests to mistakenly fail.

Unit testing is often automated at larger or more sophisticated companies.

**Integration Test**

Integration testing is a form of testing in which units of code are combined ("integrated") and the results of their interactions are tested. This type of testing is done after each of the individual units has been tested.

**Test Coverage**

"Test coverage" refers to how much of a software program has been tested. This can refer to several things, including the percentage of functions that have been called by the test suite, or the percentage of statements in the code that have been executed by the test suite.