Test - DYPL

Daniel Sundstrom
daniel@sykewarrior.com

I believe in rough consensus and running code.

March 17, 2012

1 Python vs. Ruby

There are some major differences in the philosophy underpinning the design and the choices made for each language. In short, these differences can be described as; *Multiple inheritance vs. Mixins*, *Blocks vs. Functions* and *Built-in Functions vs. Global Object*. These differences will be explored in depth below.

1.1 Multiple inheritance vs. Mixins

One of the major differences in design between Ruby and Python is how the language handles class-based behavior modification, both at runtime and at write time. Ruby favors a module-based approach, where specific methods[1] are grouped together in named modules, which in turn are included in the desired classes to provide the behavior. Modules can be included both as class methods and as instance methods, depending on the keyword used for inclusion. This is illustrated in the example below.

```
module RangedWeapons
  def attack_with_bow
    puts "#{@name} attackes with bow!"
  end
end

class Ninja
  include RangedWeapons
  def initialize(name)
      @name = name
  end
  def attack
    puts "#{@name} attacks with sword!"
  end
end
```

[1] Which, in Ruby, of course includes attributes since they are implemented as methods

```
n = Ninja.new('Shinobin')
n.attack                  # => prints "Shinobin attacks with sword!"
n.attack_with_bow         # => prints "Shinobin attacks with bow!"
```

As the example shows, this allows for composing the aggregate behavior for a class into separate, re-usable components (since a module can be included in any number of classes). Python, on the other hand, supports the same behavior through a limited form of multiple inheritance[2] as shown below:

```
class RangedNinja():
  def attack_with_bow(self):
    print self.name + " attacks with bow!"

class Ninja(RangedNinja):
  def __init__(self, name):
    self.name = name
  def attack(self):
    print self.name + " attacks with sword!"

n = Ninja("Shinobin")
n.attack()                  # => prints "Shinobin attacks with sword!"
n.attack_with_bow()         # => prints "Shinobin attacks with bow!"
```

As clearly shown in the examples above, both of these styles support the same effect - but point to interesting differences in design philosophy. One of the basic tenets of OOP[3] is its suitability and ability to map and represent real-world systems and relationships, and this is also one of basic foundation of object-oriented modeling as taught and discussed today[12][3]. The relationship of RangedNinja in regard to Ninja is more of superset-subset rather than a formal relationship. Ruby's usage of modules to support this behavior relates more closely to the concept of behavior rather than relationship. While multiple inheritance is a powerful tool[4], the presence of it in Python indicates a philosophy centered on inheritance rather than composition. The philosophy of Ruby, on the other hand, is to loosely couple shared groups of behavior with the desired classes - thus compositing the complete behavior of the class. Good example of where this is a more desired philosophy is when implementing a Data-Context-Interaction pattern.

On a larger scale this points to a difference in philosophy concerning the goals of the language, more specifically:

> "The principle of least surprise means principle of least my surprise. And it means the principle of least surprise after you learn Ruby very well. - Yukihiro Matsumoto"[10]

[2] Resolution is depth-first, left-to-right
[3] Object-Oriented Programming
[4] Though it comes at a steep price, since it increases complexity of implementation severely

Multiple inheritance is known to be quite complex to use correctly, while providing very powerful tools for specializing instance-behavior. Composition, on the other hand, is quite simple to understand and use while not being as "strict" in the computer science sense of the word.

1.2 Blocks vs. Functions

Another difference in philosophy between Ruby and Python is the treatment of closures[5]. Where Python supports closures in two major ways - lambdas and unbound functions - Ruby provides a richer set of tools for using closures - procs, lambdas, blocks and method-objects[6]. However, Python's lambdas are severely limited since they can only contain expressions and not statements, while Ruby allows any valid construct inside a closure. The difference this makes is apparent below:

```
# Ruby

[1,2,3,4,5,6,7,8,9].filter do |item|
  local_variable = SomeObject.interesting_method(item)
  return local_variable
end

# The block will have its own lexical scope and will not leak
puts local_variable # => Will raise a MethodNotFound
```

while the corresponding pythonic way of doing a filter looks something like

```
# Python

multiples_of_3 = filter(lambda x: x % 3 == 0, \
[1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Both languages support the notion of unbound functions, even though this is much simpler in Python than in Ruby (where it is actually quite complex once you need scope management), as in method objects not currently bound to an object and which can be bound at a later time to a suitable object. The main difference, however, is apparent when discussing blocks, and their nonexistence in Python. Python has an expressly stated goal from its most prolific core member that

> "Explicit is better than implicit." - Tim Peters[8]

This reflects in the choice of closure-support since the preferred way of handling them in Python - unbound methods - is a very explicit way since one decouples a bound method, making it an unbound method, passes it around (since functions are first-class objects in Python) and then binds them to a scope on execution. At all times do one act upon an Method-object. In Ruby,

[5] A closure allows a function to execute outside its immediate lexical scope

[6] It can be argued that procs and lambdas are in essence the same thing, but that falls outside the scope of this text (and is plain wrong, see the arity-handling!)

however, there's a notion of callable objects, that is - objects with respond to the method *call*. These includes all the types mentioned above (and could thus also include user-defined objects implementing the same API). One can, for example, implicitly convert a method argument into a callable object using the following syntax:

```
def block_method(&block)
    yield 'ninja'
end

block_method do |ninja_string|
  puts "Yield:#{ninja_string}"
end
```

Note how the implicitly defined block is converted into an object at method-execution and is implicitly yielded to by the method body. This allows Ruby to use closures as dynamic callbacks for other methods, something that is possible in Python but require the use of external libraries to allow higher-order functions.

Ruby supports a very implicit style of scope management for both variables, and in order to support a more functionally oriented style of programming, while Python eschews implicitly declared scopes in lieu of making the process explicit and binding it to a single object and a construct.

1.3 Built-in Functions vs. Global Object

One major design difference between Ruby and Python is how to implement and make available basic and necessary functions and methods to a default scope, i.e. the scope where code is executed unless explicitly executed in another scope. This includes *eval* and *puts* in Ruby and *file* and *print* in Python. Ruby chooses to implement these as instance-methods on a global object[7] while Python builds them into the interpreter. This has numerous implications, for example - adding new expressions in Ruby is a matter of opening up one of Object, Kernel or BaseObject (which are all present in the inheritance chain by default) and adding new instance methods - while adding new functions on that level in Python is not possible (they must be explicitly referenced). This reflects a difference in where the languages comes from. Python, which originally was intended as "[..] bridge the gap between the shell and C."[11] whereas Ruby comes from "[...] concentrate on the fun and creative part of programming when they use Ruby."[9]. Since built-in functions on an interpreter level encourages an expression-oriented style of coding (much like Lisp), something which might feel natural to someone used to C-style procedural coding, it's natural that Python should choose interpreter-level functions. Ruby, on the other hand, springing from a designer who explicitly wants to escape the then traditional style of coding has a design which encourages a completely different style of coding.

[7] Much like the window-prototype in regular Javascript

2 Reflection

Reflection (and its sibling introspection) is a term denoting the ability of an executing program to look "into" itself at runtime and influence, view and/or modify its constructs, data and business logic while still executing. The exact application of reflection differs highly from language to language, both depending on design philosophy and on technical limitations. Generally speaking, reflection ranges from the ability of a program to list, for example, properties of an object - this is the case with among others C# - to the ability to dynamically define new functions and methods, add and remove them to objects at will as well as rewriting the actual logic while executing - this is the case with among others Javascript, Ruby and Io. Dynamic method invocation is generally thought be a part of reflection as well - something which for example Java supports.

Reflection is a natural part of metaprogramming, and as such can be very useful in a wide range of situations in software development. Reflection can be used to achieve highly dynamic scriptable programs, where key scripts can be changed during runtime or programs which change their behavior in response to external input or changes in the execution environment not possible to anticipate in its entirety at program creation. A very common use of reflection is for writing polymorphic serialization routines which can work with any object, instead of forcing the programmer to write custom serialization for every type created, which is the case with the XML-serializer in Ruby Standard Library for example. Other uses of reflection includes highly configurable genetic algorithms[4] and high-level ORMs[8] which adapt methods to the underlying data (for example DataMapper and ActiveRecord).

What makes reflection useful in the cases above is the ability to make the program conform to input or the environment without having to anticipate all possible permutations of input. A good example of this is when building ORMs. The example below is using ActiveRecord, a Ruby ORM:

```
# Ruby

# A mysql-table representing Users
# +-----------------------+--------------+------+-----+
# | Field                 | Type         | Null | Key |
# +-----------------------+--------------+------+-----+
# | id                    | int(11)      | NO   | PRI |
# | login                 | varchar(40)  | YES  |     |
# | email                 | varchar(100) | YES  |     |
# +-----------------------+--------------+------+-----+

class User < ActiveRecord::Base
end

# Note that the method find_by_email is dynamically
# available based on fields in database.
```

[8] Object/Relational Mapping

```
user = User.find_by_email("daniel@sykewarrior.com")
puts user.email # => prints "daniel@sykewarrior.com"
```

The ORM reflects into the data fetched from the database, and dynamically adds methods to the User-class for the fields present, such as *email* in the example. This allows for a natural style of coding while being very adaptable since adding a new field to the database automatically will result in the corresponding methods being available, and would be very hard and complex to implement without reflection.

The biggest pitfalls with reflection is the inverse of its greatest strengths, that programs can adapt to input and the environment it executes in. Since a programmer has to define the boundaries for this adaption[9] and failure to cover the limitations properly will result in non-deterministic behavior and possibly bugs dependent on the external environment, which in turn are very hard to debug. This also makes it impossible to implement certain kinds of modifiable reflection in statically typed languages, since the compiler can't verify type checking or interface compliance at compile time. This is the reason for Java only implementing observing and invoking reflection, while for example Ruby allows for modifications to almost the entire context, program constructs and behavior[10].

An example of reflection usage is the following Ruby code which adds a "macro" for resetting classes to blank slates, since Ruby lacks any class-unloading facility.

```
class Object
 def reset(c)
  c.class_eval{ alias_method :__to_s, :to_s }
  c.instance_methods.map do |a|
    c.send(:undef_method, a) unless a.match(/^__|object_id/)
  end
  c.class_variables.map{|a| c.send(:remove_class_variable, a)}
  c.class_eval do
   def to_s
    return __to_s+"<reset>"
   end
  end
 end
end
```

This code, when run in almost any Ruby environment will add the macro[11]-style to the environment. First, it uses dynamic invocation and slot modification to store a copy of the *to_s*-method which will not be removed. Then it uses reflection to fetch all instance methods of the class-instance passed, and removes them from the class unless they either begin with __ or is *object_id* - both of which are

[9] With the exception of true generative programming, but that is at the moment more curiosity than a real issue in computer science.

[10] With the exception of class unloading which is currently impossible in Ruby

[11] This is not a true macro, but behaves macro-like.

either internal methods required by the Ruby runtime. Then it makes another pass using reflection to remove all class variables from the class and finally it uses more reflection to add a new version to *to_s* which prints the original version appended with a marker that it has been reset.

3 Typing?

In order to discuss dynamic typing, one must first discuss type errors. A type error occurs when an operation only applicable to a subset[12] of the programs available types is applied to data of the wrong type. This can either produce non-deterministic results, or result in a runtime error - both of which is in essence are type errors, though the non-deterministic result might not be treated as an error by the runtime. By not doing implicit type conversion[13] the runtime can determine if a given operation is valid for the supplied types - and, if not, raise an error.

When discussing the implications on a language resulting from dynamic typing, one must make a few important distinctions - primarily between what results thanks to dynamic typing, and what results from weak/strong typing. An example of this is prevention of applying operations to invalid types. This, while being caught by a static type checker, is not a property provided by the dynamic/static type system, it is a property of a strong type system. This distinction makes the discussion on what dynamic type systems guarantee more specific and focused.

A dynamic type system makes it possible to write and execute all possible programs. The space of possible programs is bound by the programmer, rather than the language. This is a very important distinction between static and dynamic type systems. It is generally accepted that a static type system rejects programs which, in reality, are type safe and free from type errors - it just can't be proven deterministically at compile time. A "proof" of this is that almost all statically typed languages provide features for the programmer to cast a type to some other type, with the implication that the programmer knows what he/she is doing and thus rely on this to accept the cast even though it might fail at runtime. If static type checkers were perfect, such a mechanism would not be required, since it would infer proper type anyway. If one accepts that all static type systems reject valid programs which it can't guarantee type safety for, then the implication is that dynamic type systems allow for a superset of programs, compared to static type systems. This guarantee has a number of implications on program design. If all possible programs can be written and executed, is it possible to write a program which is rejected by a dynamic type system? The answer is of course no, since it might be written and yet crash immediately upon execution. It will still be a valid program under the type system.

[12] the cardinality of which might be 1.
[13] That is, weak typing

In the same vein does a dynamic type system guarantee that all expressions permissible by the language will at least be evaluated, which is a requirement for being able to meaningfully implement modifiable reflection. If a type checker enforces constructs it can statically determine to be safe, all extensions and changes to a language need to conform to the same rules. A good example of this is game engines which rely on an embedded Lua interpreter to support plugins or custom user scripts. Should that be implemented with a language relying on static type checking, any changes to the scripting API provided by the host application would require a re-compilation of the plugin with re-evaluation of all type rules.

The term dynamic typing is a sensible term, if applied to the type system instead of the object space. The ability to dynamically change the type of objects in runtime should be cleanly separated from the ability to define new types, or change the types themselves. An example of this is the ability to define new types[14] in Ruby in runtime. This is a property of the dynamic type system, since the total set of available types are not fixed in a compiler step, while the ability to change a string to behave like an integer is a function of the ducktyping support. The term relates to the type space itself, which is dynamic during the entire execution flow (at no point is the type system frozen), in contrast to static type systems where the global set of types is frozen upon compilation. By the same account is static type a sensible term, since the global set of types is "static" once compilation has finished.

3.1 *method_missing* and static typing

Ruby supports *method_missing*, a method that is invoked on an instance whenever it is sent a message not possible to resolve and the default implementation of *method_missing* just raises a NoMethodError and returns. This is possible due to Ruby being dynamically typed, and supporting *dynamic dispatch*. At first glance, static typing seems to be incompatible with this approach due to two problems; polymorphism and operator-related type safety.

The standard implementation of *method_missing* in Ruby behaves as follows:

```
def method_missing(name, *args, &block)
  # Code
end
```

where all arguments are gathered into an array, and any passed block are stored in *block*. *name* is the name of the method invoked but not found. This allows the instance to, for example, perform introspection using data derived from the passed name to determine how to handle the invocation, or call *super* and pass it on upwards in the inheritance chain. In order to make a similar implementation compliant with a static type system, the system needs to be able to make certain guarantees on the behavior of the implementation. It is not possible to add an implementation like the one in Ruby to a statically typed language, since there

[14] i.e. classes

is no notion on what type(s) are returned, or which ones are passed to the invoked method. One could however design an implementation that could, in theory, comply with a static type checker.

Using Java[15] as an example (or in reality, any language supporting *ad-hoc polymorphism*) one could define an implementation of *method_missing* for each supported type signature. An pseudo-example is included below:

```
public MethodMissing{
    public String method_missing(String name,
                                 ArgumentList<String, Integer, String> args)
    {
        // Implementation
    }
    public Integer method_missing(String name,
                                  ArgumentList<Integer, Integer> args)
    {
        // Implementation
    }
}
```

The pseudo-code above introduces a new basic type called *ArgumentList* which is parametrically typed to an ordered list of method arguments. The compiler would then match the implementation with a matching signature or throw an error. The return value can be checked in much the same way.

The effects of this with regard to static typing is that in order to properly type check any implementations providing this functionality, the implementations themselves needs to carry enough information (in essence, providing an interface) to resolve. This makes the implementations themselves so narrow (in terms of applicability) that they provide limited additional value. This is probably the reason for this not being implemented in many statically typed languages already, as it can be provided using other means.

Another effect is its limited scope of usage. Since a statically typed language will infer the correct method mapping at compile time (or the range of applicable methods in a lookup table) the actual scope where *method_missing* applies is restricted to invocations resulting from reflection based on dynamic input. While this is a valid case in itself, one could argue that there's no difference in that case from using reflection to determine if the instance responds to the selector at hand and, if not, take some other action (like throwing an error).

4 Scripting?

There is no standard definition of what denotes a scripting language, however few themes are recurring and commonly shared between languages agreed upon

[15]It should however be noted that this can already be achieved in Java using proxies instead.

to be scripting languages. Firstly, the lack of an explicit compiling and linking step. Contrary to compiled languages which require one or more compiling steps scripting languages are typically interpreted and executed on fly, passing the raw source code to the interpreter. Secondly, it is rare for programs written in a scripting language to be distributed in any other form than source, while compiled languages can be distributed as their binary, compiled products[16]. Other common traits of scripting languages is the lack of manually managed memory, usually relying on garbage collection with either reference counting (like Perl or Python) or a mark-and-sweep strategy (like Ruby) - though this is becoming present in non-scripting languages as well (most notable Obj-C with ARC[17]). Thirdly - scripting languages tend to be used more for prototyping, rapid development, agile methods and similar development projects due to the flexible and less rigid nature of the code produced. This is by no means an absolute, but there's a strong tendency towards it. Since scripting languages lack a compiling/linking step, compile time type-checking is not seen in these kinds of languages[18].

For the purpose of this essay, a scripting language shall be defined as any *language not required an explicit compilation and/or linking steps before executing.* Languages falling outside this definition shall also include languages compiling to bytecode (such as Java or ActionScript) before being distributed or run. Languages which are defined as scripting languages include, but are not limited to Ruby, Javascript, PHP, BASH, Perl and Python.

Literals, which is a language construct for representing values and/or objects by implicitly creating them on execution, provide programmers with a "lighter" way of initializing objects. This can range from strings (which are commonly available as literals)

```
// Objective-C

NSString *str = @"a string";
```

to more complex datatypes:

```
# Ruby

structure = {:array => [1,2,3], :string => "ninjas", :fixnum => 4}
```

to functions:

```
-- Lua

fact = function(x)
  if 0 == x then
    return 1
```

[16]Though the reverse is not strictly true as is very common for open-source packages to be distributed as source and compiled on the target platform.

[17]Automatic Reference Counting

[18]Though this becomes a little fuzzy since among others, the release of Diamondback Ruby which provides static typing to Ruby

```
  else
    return x * fact(x-1)
  end
end
```

The presence of literals in a scripting language is important because of three major implications; shorter REPL[19], higher abstraction-level and "*mockability*".

4.0.1 Shorter REPL

As stated above, scripting languages are often employed for rapid development and prototyping where basic tenets programs are built on can rapidly change - thus the ability to rapidly change data structures or add new one becomes an important feature. The possibility to change add an array to a method, or change a map to an array provides quicker feedback on the written code, since the time between REPL-interations becomes shorter. This in turn frees the programmer from having to plan out the entire program before starting coding, thus increasing the feedback-loop between thinking, coding and testing. The benefits of this have been thoroughly explored in the TDD[20] community[6][7] since the general consensus is that while test-driven development takes longer time, end result is of higher quality. Thus, shorter REPL leads to higher productivity, both when using TDD and when using other development strategies.

4.0.2 Higher Abstraction

Consider the following example:

```
# Ruby

# If run on 32-bit system, use 2**30-1
number = 2**62-1
puts number.class # => returns Fixnum

number += 1
puts number.class # => returns Bignum
```

The code above uses number literals to create a variable *number* in which it first stores a very large number[21] and then prints its class. The class returned is Fixnum, which represents an integer value small enough to store in a machine word[22]. When the integer increases past the size possible to store in a Fixnum, the interpreter takes the literal and automatically uses the class Bignum instead (which uses multiple machine words to store integers, thus increasing the limit significantly). The programmer does not have to reflect on whether the variable in question can grow larger than given limits and can thus focus on writing code rather than managing that kind of special cases.

[19] Read-Eval-Print-Loop
[20] Test-Driven Development
[21] 4611686018427387903 to be exact
[22] A machine word minus 1 bit since 1 bit is used to mark reference vs. integer

4.0.3 Mockability

When developing a program in languages requiring explicitly typed data structures, the programmer often has to write significant parts of the code before it is possible to test the program flow, verify its usability or run it against test data to verify algorithms. Since the language requires the programmer to define exactly what type of objects will inhabit a given structure, those types has to be defined and be at least partially implemented before execution flow can move past the creation of the data structure in question. This prohibits testing of the entire logical flow before significant parts of the program is already written, and it limits the options of experimenting while analyzing the problem solved by the program. This also implies that the cost-of-change for a making a given decision can be very high, and have far-reaching implications. An untyped, literal-based creation of data structures allows for implementing the smallest part required for testing the purpose of the program, the business logic and usability - in turn lowering cost of development, increasing possibility of semantic correctness of the program and increasing developer creativity in problem solving.

5 Weak/Strong vs. Dynamic/Static?

According to Craig[1] it is impossible to discuss programming languages without discussing their type systems. He relates this to the fact that

> [...] the untyped λ-calculus has no consistent models, thus demonstrating that untyped languages produce unpredictable results [...]

thus any language producing predictable results needs to have some kind of type system. Two of the major concept pairs in type theory are weak/strong typing and dynamic/static typing. These represent different approaches to adding a type system, and have very different implications for both language design and compiler/interpreter-design.

5.0.4 Weak / Strong typing

Weak typing implies that the type system supports implicit type conversion by the runtime environment, and that methods usually don't type-check parameters. Strong typing on the other hand implies that types might only interact with other types in clearly specified ways, such as one cannot subtract strings from integers or multiply arrays. Both of these concepts are strongly related to type safety - that is, the ability to guarantee that operations intended for one type does not inadvertently operate on any other type which might produce non-deterministic results. An example of weakly typed language is Javascript, which the example below indicates:

```
// Javascript

var variable = "2";

console.log(variable*3); // => 6 (Number)
```

```
console.log(variable+3); // => "23" (String)

console.log((variable+3).length);
// => 2 (Number) (since the lenght of string "23" is 2)
console.log((variable*3).length);
// => undefined (since length of number 6 is undefined)
```

Note that the operation *+* performs an implicit type conversion on the integer *3*, casting it as String from Number, while the operation * reverses the operation, instead converting the string *"2"* into an instance of Number with value *2* and producing a wildly different result. Unless the programmer knows the rules of this implicit conversion this can lead to unexpected results further on (as the example proves). One should note, however, that this behavior allows for using *ad-hoc polymorphism* which can yield powerful levels of freedom in program design. This kind of operation would not be allowed in strongly typed language, such as Java, as shown in the example below:

```
public class StrongType {
    public static void main(String args[]){
        Float num = new Float(2.0);
        String var = new String("2");
        System.out.println(var * num);
        // throws an Exception "Unresolved compilation problem"
    }
}
```

Note that the compiler doesn't even accept compilation, much less execution of the program, since the behavior of * is not defined for arguments of type String and Float. This prevents the program yielding the kind of unexpected results showcased in the Javascript example above.

5.0.5 Dynamic / Static typing

The concepts of dynamic and static typing relate to how/when/if type checking is done. Type checking is the process of validating that the operations performed are compatible with the types it is performed on. This can either be done at compile time (static type) or at runtime (dynamic type)[23]. An example of a statically typed language is Objective-C, as shown here:

```
#import <Foundation/Foundation.h>
int main (int argc, const char * argv[])
{
  @autoreleasepool {
    int m = 3;
    char string[6] = "string";
    NSLog(@"%i", m*string);
  }
  return 0;
}
```

[23]Though C++ makes an interesting example where it can done at both times through the type information available at runtime.

This example will not compile, the compiler will throw an error stating that *Invalid operands to binary expression ('int' and 'char *')* and cancel the compilation. Change the type of *string[]* to an *int* would let the compilation finish. Thus, type checking is performed at compile time, making Objective-C a statically typed language. Language with dynamic typing performs the type check at runtime, and will throw a runtime error if an operation is applied to incompatible types. This can be described as values having types but variables are untyped. This allows for dynamic creation of types, for example - creating new classes in runtime in Ruby, which is a dynamically typed language. The example below displays this:

```
class Ninja
  def initialize(name);@name = name;end
end
ninja = Ninja.new("Shinobin")
puts ninja / 2
# => Will throw a NoMethodError, since the type Ninja is
# not compatible with division operations
class Ninja
  def /(args)
    return @name.length / args
  end
end

puts ninja / 2
# => prints 4, since type Ninja now supports
# division operations
```

Note that no type-checking is performed until the operation is actually performed (which fails)[24], then re-evaluated the next time at which point the type *Ninja* supports division (albeit with a quite weird implementation) and a proper result is returned. This makes Ruby a dynamically typed language, although this particular style of dynamic typing is often referred to as *Duck Typing* since it's not really the type being checked, it's the conformity of the object to the requested interface (in this example, support of division operations). The term implies that the actual identity of the object is irrelevant, and that the relevant issue is whether it can perform the requested operation, as stated by Alex Mertelli[5], credited with inventing the term in its current usage:

> In other words, don't check whether it IS-a duck: check whether it QUACKS-like-a duck, WALKS-like-a duck, etc, etc, depending on exactly what subset of duck-like behaviour you need to play your language-games with.

References

[1] I. Craig. *The Interpretation of Object-Oriented Programming Languages 2 ed.* Springer, 2002.

[24] For the sake of clarity in the example, the rescue-clause required here has been removed.

[2] S. Isoda. Object-oriented real-world modeling revisited. *Journal of Systems and Software*, 59(2), 2001.

[3] M. Keijzer, U.-M. O'Reilly, S. M. Lucas, E. Costa, and T. Soule, editors. *Genetic Programming, 7th European Conference, EuroGP2004, Coimbra, Portugal, April 5-7, 2004, Proceedings*, volume 3003 of *Lecture Notes in Computer Science*. Springer, 2004.

[4] A. Mertelli. Re: Type checking in python?, Jul 2000.

[5] NRC Institute for Information Technology. *On the Effectiveness of Test-first Approach to Programming, Proceedings of the IEEE Transactions on Software Engineering*, volume 31(1). NRC Institute for Information Technology, 2005.

[6] F. Padberg and M. Müller. About the return on investment of test-driven development. In *ICSE-Workshop on Economics-Driven Software Engineering Research EDSER*, pages 26–31, May 2003.

[7] T. Peters. The zen of python, Aug 2004.

[8] B. Stewart. An interview with the creator of ruby, Nov 2001.

[9] B. Venners. The philosophy of ruby - interview with yukihiro matsumoto, Sep 2003.

[10] B. Venners. Python's design goals - interview with guido van rossum, Jan 2003.

[11] N. Wirth. Good ideas, through the looking glass. Technical report, Swiss Federal Institute of Technology - Department of Computer Science, 2005.