Advanced Topics in Programming Language - Duck Typing in Python

Tommaso Puccetti
Studente presso Universita degli studi di Firenze

May 11, 2019

Contents

List of Tables

List of Figures

1 Possible references

- You Tube video;
- Python duck typing (or automatic interfaces)- how change the depen-decy injection;
- Simple example;
- Something more technical;
- Ultimate guide to Python's type checking;
- Static vs Dynamic (Stack Overflow)
- Why Python is dynamic and strongly typed
- Maybe schematic overview on typechecking
- Bound methods vs Function
- Annotation power in Python

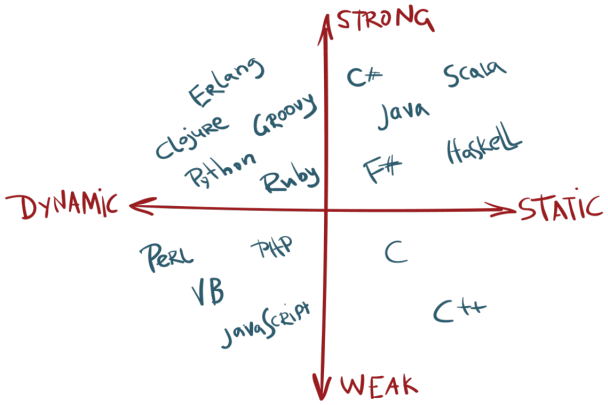
2 Questions

- Are language that implement type inference always static ?

3 Type checking: classification

Type checking is the process of verifying and enforces the typing rules of a language. In other words the **type checker** (the type checking algorithm of the language) is used to prove the **type safety** of a program.
Here the possible categories:

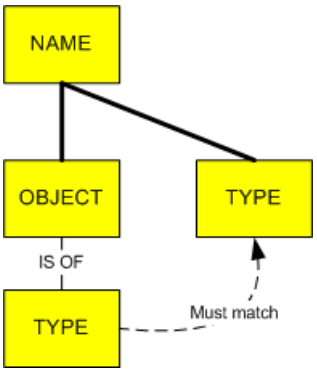
- Dynamic vs. Static;
- Explicit vs Implicit;
- Weakly vs Strongly



3.1 Static type checking

Is the process of verifying the type safety of a program based on the analysis of a program text. If a program passes a static type checker, then the program is guaranteed to satisfy some set of type safety properties for all possible inputs ***(Wikipedia)***.
A language is statically typed if the type of a variable is known at compile time. For some languages this means that you as the programmer must specify what type each variable is (e.g.: Java, C, C++); other languages offer some form of type inference, the capability of the type system to deduce the type of a variable ***(Stack Overflow)***;

A language is statically-typed if the type of a variable is known at compile-time instead of at run-time. Common examples of statically-typed languages include Java, C, C++, FORTRAN, Pascal and Scala. ***(Schematic overview)***;



3.1.1 Example

Here a java example:

```
int variable;
variable = 10;
variable = "ten";
```

It causes a compilation error:

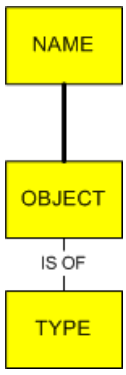
```
INSERIRE ERRORE DI COMPILAZIONE
```

3.1.2 Advantages

- A large class of errors are caught in the early stage of development process;
- Static typing usually results in compiled code that executes more quickly because when the compiler knows the exact data types that are in use, it can produce optimized machine code.

3.2 Dynamic type checking

Is the process of verifying the type safety of a program at runtime. It may cause a program to fail at runtime ***(Wikipedia)***.
A language is dynamically typed if the type is associated with run-time values, and not named variables/fields/etc. This means that you as a programmer can write a little quicker because you do not have to specify types every time (unless using a statically-typed language with type inference) ***(Stack Overflow)***.
In Dynamically typed languages, variables are bound to objects at run-time by means of assignment statements, and it is possible to bind the same variables to objects of different types during the execution of the program. Dynamic type checking typically results in less optimized code than static type checking. It also includes the possibility of run time type errors and forces run time checks to occur for every execution of the program (instead of just at compile-time) (***Schematic overview***).



3.2.1 Example

```
variable = 10
variable = "ten"
```

3.2.2 Advantages

- Implementations of dynamically type-checked languages generally associate each run time object with a type tag (i.e., a reference to a type) containing its type information. This run-time type information (RTTI) can also be used to implement dynamic dispatch, late binding, down-casting, reflection, and similar features;
- The absence of a separate compilation step means that you dont have to wait for the compiler to finish before you can test your code changes. This makes the debug cycle much shorter and less cumbersome.

CAPIRE SE SONO SOTTO CATEGORIE DI STATIC O DISTINZIONE PIU GENERALE

3.3 Explicitly typed

Each variables is annotaded in source code with type's information. In this case the *type check is simple but the language is more difficult* (from the programmers point of view).

3.4 Implicitly typed (inference)

The data types of source code are automatically detected. It is also rederred as **type inference**. The language *is easier but the type check algorithm is far more complex*.

3.5 Strongly typed

A strongly-typed language is one in which variables are bound to specific data types, and will result in type errors if types do not match up as expected in the expression regardless of when type checking occurs. (***Schematic overview***)

3.5.1 Example

```
variable1 = 10
variable2 = "ten"
variable3 = variable1 + variable2
```

3.6 Weakly typed

A weakly-typed language on the other hand is a language in which variables are not bound to a specific data type; they still have a type, but type safety constraints are lower compared to strongly-typed languages.

```
$temp = "ten";
$temp = $temp + 10; // no error caused
echo $temp;
```

3.7 Example: type inference vs. dynamic typing

These two kind of typings could be confused. Here an example to clarify the differences:

```
var1 = 10
var2 = "astring"
var3 = var1 + var2
```

1. In **dynamically typed** language this code run without errors: at runtime the *var1* is forced to be a string and the result is *"10astring"*;
2. By the other side, in **inferred type language** the compiler *throw an error.*

4 Python type checking

4.1 Dynamic and strongly typed

Python is **dynamic**: objects have a type but it is determined at runtime.
It rarely uses what it knows to limit variable usage.

```
if False:
    print(10+"ten")
else:
    print(10+10)
```

The first branch never execute, so the type checking ignore the type incongruency.
Let's try to run:

```
print(10+"ten")
```

Once executed the type check raise a type error:

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

There are some operations (methods, we will see later introducing duck typing) used even in case of type incongruence.
The boolean equivalence is permitted in Python 2 and 3:

```
print("10" == 10)
print("10" != 10)
```

Returning:

```
False
True
```

In Python 2 "*grather than*" and *"less than"* are permitted:

```
print("10">10)
```

```
print("10">=10)
print("10"<10)
print("10"<=10)
```

Returning:

```
True
True
False
False
```

Python 3 do not allowed to do "*grather than*" and *"less than"* controls.

Another consequnce is that programmers are **free to bind the same names (variables) to different objects with a different type**. Then the following statements are perfectly legal:

```
variable = 10
variable = "ten"
```

So long as you only perform operations valid for the type the interpreter doesn't care what type they actually are.

Python is also **strongly typed** as the interpreter keeps track of all variables types. In Python is not allowed to perform operations inappropriate to the type of the object: attempting to add numbers to strings will fail.

One of the advantage of the **strongly typing** for the **dynamic** language is that *you can trust what's going on:* if you do something wrong, your program will generate a type error telling you where you went wrong, and you don't have to memorize a lot of arcane type-conversion rules or try to debug a situation where your variables have been silently changed without your knowledge.

4.2 Duck typing

4.2.1 Object oriented

The following code show how to declare a class, define a costructor for its parameters, define others methods and access to both.

```
class Duck():
    #Constructor
    def __init__(self, name, colour):
        self.name = name
        self.colour = colour

    def quack(self):
        return "Quaaack"

    def fly(self):
        return "The duck is flying"

#Instantiate a istance of the Duck() class
donald = Duck("Donald","white")

#Access to fields and methods

print(donald.name)
print(donald.colour)

print(donald.quack())
print(donald.fly())
```

```
Donald
white
Quaaack
The duck is flying
```

The first argument of every class method, including init, is always a reference to the current instance of the class. By convention, this argument is always named *self*. In the init method, self refers to the newly created object; in other class methods, it refers to the instance whose method was called.
The *self* world is the equivalent of *this* in **Java**. However Java do not requires to pass *this* explicitly as a first parameter of a method, it could be used straight in the body of the method for its pourpouse. However self is not a reserved keyword in Python its just a strong convention. We could rewrite the previous class as follows:

```
class Duck():
    #Constructor
    def __init__(myself, name, colour):
        myself.name = name
        myself.colour = colour
```

```
    def quack(myself):
        return "Quaaack"

    def fly(myself):
        return "The duck is flying"
```

In Python is not possible to define multiple constructor for a class, still is possible to define a default value if one is not passed.

```
class Parrot():
    def __init__(self, name = "Perry"):
    self.name = name

bird1 = Parrot()
bird2 = Parrot("Jack")

print(bird1.name)
print(bird2.name)
```

```
Perry
Jack
```

4.2.2 Main idea

Duck typing is a concept related to dynamic typing in an object oriented language. Is a feature in which the semantics of a class is determined by its ability to respond to some message (method or property) rather than being the extension of a class or an implementation of an interface.

If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck.

```
    class Duck():
        def quack(self):
            return "Quaaack"
        def fly(self):
            return "The duck is flying"

    class Parrot():
        def quack(self):
            return "The parrot parrots a quack"
```

```
    def fly(self):
        return "The parrot is flying"

class Man():
    def quack(self):
        return "The man parrots a quack too"

v = [Duck(), Parrot(), Man()]

for i in v:
print(i.quack())
```

The idea is that it doesn't actually matter what type my data is - just whether or not i can do what i want with it.

```
for i in v:
    print(i.fly())
```

If we try to use the method *fly()* over the entire collection of objects an error is raised at runtime:

```
Traceback (most recent call last):
File "/home/tommaso/git/ducktyping-tpl/code/ducklist.py", line 23, in <module>
print(i.fly())
AttributeError: Man instance has no attribute 'fly'
```

Lets see another example:

```
class Car:
    def __init__(self, engine):
        self.engine = engine
    def run():
        self.engine.turn_on()
```

This is a classical example of **dependency injection**. My class Car receives an instance of an engine and use it in the run method, where it calls the *turn_on* method. Note that my Car does not depends on any concrete implementation of engine. And I'm not importing any other type! Just using a dependency injected instance of something that responds to a *turn_on* message. I could say my class Car depends on an interface. But I did not have to declare it. **It is an "automatic interface"!**

In a language **without** duck typing I'll probably have to declare an explicit interface named for example *IEngine*, have the implementation (for example *EngineV8*) and explicit define my Car parameter to be an implementation of IEngine.

```
interface IEngine {
    void turnOn();
}

public class EngineV8 implements IEngine {
    public void turnOn() {
        // do something here
    }
}

public class Car {
    public Car(IEngine engine) {
        this.engine = engine;
}

public void run() {
    this.engine.turnOn();
```

Another intresting feature in Python, due to the duck typing, is the chance to add methods to the classes. First of all is important to say that in Python there is ad difference beetween:

- **Function**;
- **Bound method**.

Bound methods have been "bound" to an instance, and that instance will be passed as the first argument whenever the method is called.

```
>>> def foo():
...     print "foo"
...
>>> class A:
...     def bar( self ):
...         print "bar"
...
>>> a = A()
>>> foo
<function foo at 0x00A98D70>
>>> a.bar
<bound method A.bar of <__main__.A instance at 0x00A9BC88>>
>>>
```

Callables that are attributes of a class (as opposed to an instance) are still unbound, though, so you can modify the class definition whenever you want.

Recalling the previous example of calling a method on a list of object we have the following behaviour:

```
donald = Duck()
charlie = Parrot()
john = Man("John")
jack = Man("Jack")
v = [donald, charlie, john, jack]

def fly(self):
    return "Takes a plane"

Man.fly = fly

for i in v:
    print(i.fly())
```

The fly method was added to all the instances of Man class so that, when the fly method is called on the list there, are no errors at runtime.