

# Advanced Topics in Programming Language - Duck Typing in Python

Tommaso Puccetti  
Studente presso Università degli studi di Firenze

May 6, 2019

## Contents

<b>1</b>	<b>Possible references</b>	<b>2</b>
<b>2</b>	<b>Questions</b>	<b>2</b>
<b>3</b>	<b>Type checking: classification</b>	<b>3</b>
3.1	Static type checking . . . . .	3
3.1.1	Example . . . . .	4
3.1.2	Advantages . . . . .	4
3.2	Dynamic type checking . . . . .	5
3.2.1	Example . . . . .	5
3.2.2	Advantages . . . . .	6
3.3	Explicitly typed . . . . .	6
3.4	Implicitly typed (inference) . . . . .	6
3.5	Strongly typed . . . . .	6
3.5.1	Example . . . . .	6
3.6	Weakly typed . . . . .	7
3.7	Example: type inference vs. dynamic typing . . . . .	7
<b>4</b>	<b>Python type checking</b>	<b>8</b>
4.1	Dynamic and strongly typed . . . . .	8
4.2	Duck typing . . . . .	9
4.2.1	Main idea . . . . .	9

## List of Tables

## List of Figures

### 1 Possible references

- You Tube video;
- Python duck typing (or automatic interfaces)- how change the dependency 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

### 2 Questions

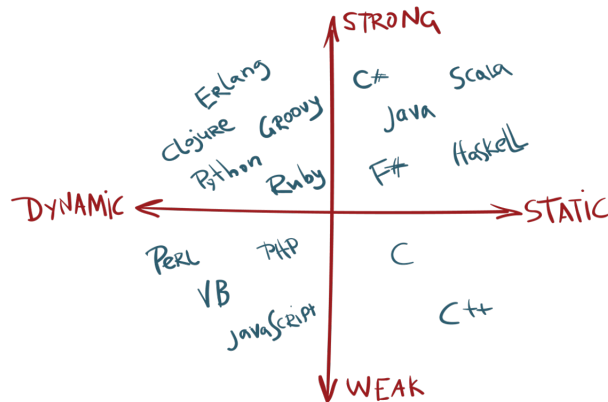
- Are language that implement type inference always static ?

### 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

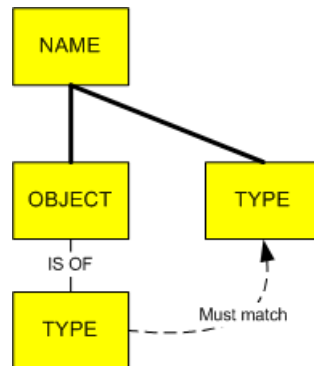


#### 3.1 Static type checking

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

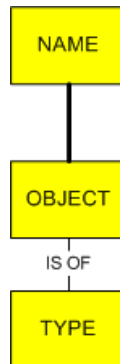
- A large class of errors are caught in the early stage of development process;
- Static typing usually results in compiled code that executes more quickly because when the compiler knows the exact data types that are in use, it can produce optimized machine code.

## 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 don't 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 variable is annotated in source code with type's information. In this case the *type check is simple but the language is more difficult* (from the programmer's point of view).

## 3.4 Implicitly typed (inference)

The data types of source code are automatically detected. It is also referred 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 run-time 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'
```

Another consequence 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 Main idea

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

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.

```
class Duck:
    def quack(self):
        print("Quaaaack")

class Man:
    def quack(self):
        print("Imitates a quack")

class Eagle:
    def fly(self):
        print("Dude i do not quack, i just fly")

class MakeItQuack:
    def __init__(self, animal):
        animal.quack()

MakeItQuack(Duck())
MakeItQuack(Man())
MakeItQuack(Eagle())
```

```
Quaaaack
Imitates a quack
Traceback (most recent call last):
  File "/Users/gaurav/personal/development/grasp-python/base/advanced/duck_typing.py", line 33, in
    MakeItQuack(Eagle())
  File "/Users/gaurav/personal/development/grasp-python/base/advanced/duck_typing.py", line 29, in
    animal.quack()
AttributeError: 'Eagle' object has no attribute 'quack'
```

This is a classical example of *dependency injection*.  
My class **MakeItQuack** receives an instance of an **animal** and use it to do something. Note that my MakeItQuack instances does not depends on any concrete implementation of animal. And I'm not importing any other type! Just using a dependency injected instance of something that responds to a quack message. I could say my class MakeItQuack depends on an interface. But I did not have to declare it. **It is an automatic interface!**