

# Duck Typing in Python

Author: Tommaso Puccetti

Università degli Studi di Firenze

21/12/2018



# Index

# Introduction

Python is an ***interpreted, multi-paradigm*** language. It was initially designed by Guido van Rossum in 1991 and developed by Python Software Foundation. It supports:

- **Functional programming** (non pure);
- **Procedural programming**;
- **Objected oriented**.

# Python's semantic

Could be useful to first recall the difference between **strict** and **lazy** evaluation:

- ➊ **Strict evaluation strategy:** the arguments of a function are fully evaluated to values before evaluating the function call (call by value);
- ➋ **Non-strict or Lazy evaluation:** arguments are evaluated only if it is needed in the function body (*call by name*)

Python:

- implements **strict semantic**;
- uses **whitespace indentation**, rather than curly brackets or keywords, to delimit blocks.

# Semantic: Python vs Haskell

In Python we never get *true* because he forced the evaluation of the function wich is an infinite loop:

```
def infiniteLoop(x):  
    while True:  
        print("do something with x")  
    return x  
  
5 in [5, 10, infiniteLoop(5)]
```

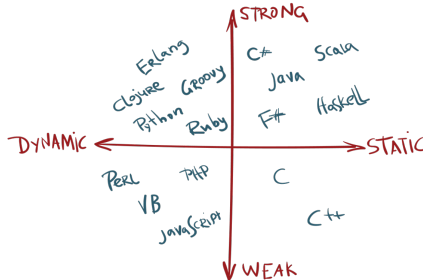
If we write the same code in **haskell** we get the *true* value:

```
elem 2 [2, 4, noreturn 5]
```

# Type checker (1)

**Type checking** is the process of verifying and enforces the typing rules of a language.

- 1 **Dynamic vs. Static**
- 2 **Weak vs Strong.**



# Type checker (2)

## 1 Dynamic vs. Static

- **Statically-typed languages:** typechecking is done at compile-time, in order to guarantee the absence of run-time (type) errors: formal proof of type-safety.
- **Dynamically-typed languages:** dynamic type checking is the process of verifying type constraints at runtime, during execution.

## 2 Weak vs Strong

- AGGIUNGERE STRONGLY
- AGGIUNGERE WEAKLY

# Python's type checker

## ① Python is **dynamic**:

- objects have a type but it is determined at runtime;
- variables are not explicitly typed
- an assignment binds a name to an object anche the object could be of any type

## ② Python is also **strongly typed**.

Let's see the implications by some example.



# Python's dynamic typing example (1)

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

The first branch never execute, so the type checking ignore the type incongruency.

If we try to execute **separately** the first branch, the type check raise a type error:

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

## Python's dynamic typing example (2)

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's strong typing example

Python is not allowed to perform operations inappropriate to the type of the object:

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

In a **weakly-typed** language, like PHP, the integer is forced to be a string and no type error is raised:

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

The output will be "ten10".

# Some exceptions (1)

There are some operations allowed 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
```

## Some exceptions (2)

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

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

Returning:

```
True  
False
```

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