

# Duck Typing in Python

Author: Tommaso Puccetti

Università degli Studi di Firenze

21/12/2018



# Index

- ① Introduction
- ② Python's semantic
- ③ Type-checking
- ④ Python Type checking
  - Dynamic typing
  - Strong typing
  - Annotations
- ⑤ Object oriented
- ⑥ Duck typing
  - Main idea
  - Dependency injection
  - Adding method to a class
- ⑦ Conclusion

# 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 evaluation**;
- uses **whitespace indentation**, rather than curly brackets or keywords, to delimit blocks.

# Strict semantic: example

In Python we never get *true* because it force the evaluation of the function wich contains an infinite loop in the body:

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

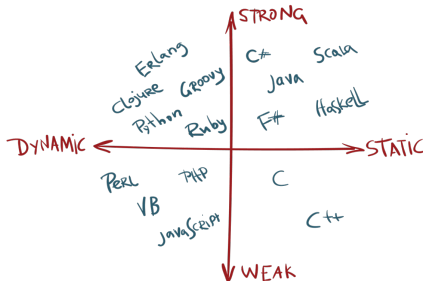
If we write the same code in **haskell** we get the *true* value:

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

# Type-checking

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

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



# Type-checking: static

**Statically-typed languages:** type-checking is done at compile time, in order to guarantee the absence of run-time errors.

- **Advantages:**

- There is a formal proof of **type-safety**;
- A large class of errors are caught earlier;
- Types guide code development;
- Types could be seen as documentation for the code.

- **Disadvantages:**

- Static typing is a constraint on the program structure;
- More code.

# Type-checking: dynamic

**Dynamically-typed languages:** dynamic type checking is the process of verifying type constraints at runtime, during execution.

- **Advantages:**

- These languages are more flexible;
- less code.

- **Disadvantages:**

- Programs can fail at runtime due to type errors.
- It forces runtime checks to occur for every execution of the program, at any step of evaluation. The result is **less optimized** code.



# Type-checking: strong and weak

- 1 **Strongly typed:** every expression is well typed;
  - **Advantages:**

# Python's type checking

## ① Python is **dynamic**:

- Variables have no type, only the object that a variable references has a type. **Variables are simply names pointing to objects**;
- variables are not explicitly typed;
- objects have a type but it is determined at runtime;

## 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 'st.r.'
```

## 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".

# Annotations

Annotations were introduced in Python 3.0 and are the main way to add type hints to the code. We can annotate both **function** and **variable**.

```
import math

pi: float = 3.142

def circumference(radius: float) -> float:
    return 2 * math.pi * radius
```

Type hints and annotations ***do not add a real static typechecking*** in native Python so this should not effect the code performance.

# Annotations: why use it?

## From PEP 484:

*" <...>using type hints for performance optimizations is left as an exercise for the reader".*

## Advantages:

- Type hints help document your code;
- Type hints improve IDEs and linters. This allows IDEs to offer better code completion and similar features.

## Disadvantages

- Type hints take developer time and effort to add.
- Type hints introduce a slight penalty in start-up time.

# Object oriented (1)

```
class Duck():  
    def __init__(self, name, colour):  
        self.name = name  
        self.colour = colour  
    def quack(self):  
        return "Quaaack"  
    def fly(self):  
        return "The duck is flying"
```

```
donald = Duck("Donald","white")
```

```
donald.name  
donald.colour  
donald.quack()  
donald.fly()
```



## Object oriented (2)

- The first argument of every class method is always a reference to the current instance of the class (***self***).
- The ***self*** word 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.
- However **self** **is not a reserved keyword** in Python, is just a strong convention.

## Object oriented (3)

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)
```

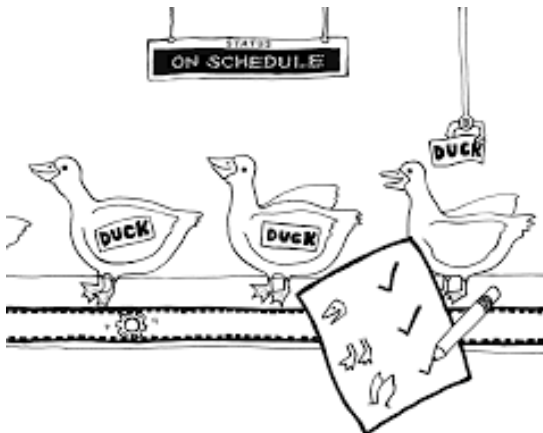
The output would be:

"Perry"

"Jack"

# Duck typing

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



# Duck typing: main idea

**Duck typing** is a concept related to dynamic typing in an object oriented language:

- Using duck typing you do not check types at all. Instead you check for the presence of a given method or attribute.
- **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.**

# Duck typing: example (1)

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

class Parrot(Bird):
    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"
```

## Duck typing: example (2)

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

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

Even if the instance of Man is not a subtype of the Bird class the type-checker do not raise any type error. The output would be:

```
Quaaaack
The parrot parrots a quack
The man parrots a quack too
```

## Duck typing: example (3)

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

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

```
The duck is flying  
The parrot is flying  
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'
```

# Duck typing: dependency injection (1)

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

- This is a classical example of **dependency injection**;
- Note that my Car does not depends on any concrete implementation of engine: i'm 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.



## Duck typing: dependency injection (2)

In a language **without** duck typing, like Java, is necessary to declare an explicit interface (*IEngine*), its implementation (*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();  
    }  
}
```

# Add method to a class

In Python there is a difference between:

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

```
>>> 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>>
```

# Add method to a class: example (1)

However is possible to poke a function into a class:

```
class Man():
    def __init__(self, name):
        self.name = name
    def quack(self):
        return "The man parrots a quack too"

donald = Duck()
charlie = Parrot()
john = Man("John")
jack = Man("Jack")

v = [donald, charlie, john, jack]
```

## Add method to a class: example (2)

```
def fly(self):  
    return "Takes a plane"  
  
Man.fly = fly  
  
for i in v:  
    print(i.fly())
```

Every instance of the Man class, even if previously instantiated, now has the fly method.

```
The duck is flying  
The parrot is flying  
John takes a plane  
Jack takes a plane
```

# Add method to a single instance of a class (1)

It is possible to add a method to a single instance of a class but we have a problem: **the function is not automatically bound** when it's attached directly to an instance.

```
john.fly = fly

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

```
The duck is flying
The parrot is flying
Traceback (most recent call last):
File "/home/tommaso/git/ducktyping-tpl/code/pokelist.py", line 33,
in <module> print(i.fly())
TypeError: fly() takes exactly 1 argument (0 given)
```

## Add method to a single instance of a class (2)

To properly bound the method to "john" we had to use the module *types*:

```
import types

john.fly = types.MethodType(fly, john)

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

## Add method to a single instance of a class (3)

We still have an error but this time is caused by the instance "jack", proving that we added the method fly only to one instance of Man.

```
The duck is flying
The parrot is flying
John takes a plane
Jack takes a plane
Traceback (most recent call last):
File "./.home./.tommaso./.git./.ducktyping.-.tpl./.code./.pokelist...py.",
line 35, in <module>
print(i.fly())
AttributeError: Man instance has no attribute 'fly'
```

# Conclusion