

# Duck Typing in Python

Author: Tommaso Puccetti

Università degli Studi di Firenze

21/12/2018



# Index

- ① Introduction
- ② Python evaluation strategy
- ③ Type-checking
- ④ Python type-checking
  - Dynamic typing
  - Strong typing
  - Annotations
- ⑤ Object oriented
- ⑥ Duck typing
  - Main idea
  - Duck typing vs. Static typing
- 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 ;**
- **Procedural programming;**
- **Objected oriented.**



# Python's evaluation strategy

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

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

**Python:**

- implements **strict evaluation**.

# Strict evaluation: 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)]
```

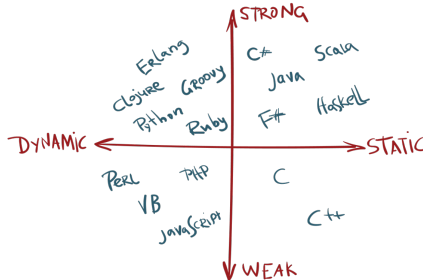
If we write the same code in **haskell** we get the *true* value:

```
elem 5 [5, 10, infiniteLoop 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

In **statically-typed languages** type-checking is done at compile time.

- **Advantages:**

- There is a formal proof of **type-safety** which guarantee the absence of run-time errors.;
- Static typing 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

In reverse, in **dynamically-typed languages** type checking take place at run-time, during the execution of the code.

- **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. The result is **less optimized** code.



# Type-checking: strong and weak

- 1 In a **strongly typed** language every sub-term must be well typed, which means that the language enforces strict restrictions on mixing of values with differing data types in an expression.
- 2 In a **Weakly typed** language there are no such restriction.

# Python type-checking

- 1 Python type-checking is **dynamic**:
  - **Variables are simply names pointing to objects**, only the object that a variable references has a type;
  - objects have a type but it is determined at runtime;
  - variables are not explicitly typed;
- 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, having different types**. 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 they do 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

```
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()
```

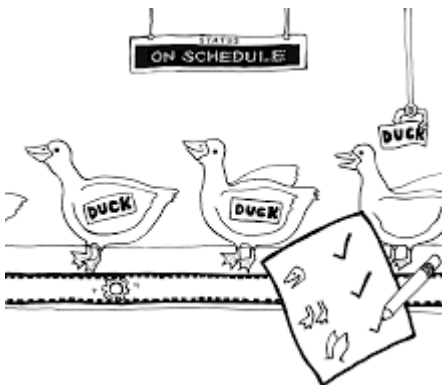


# Object oriented: self

- 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.
- However self **is not a reserved keyword** in Python, is just a strong convention.

# 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:

- Implementing duck typing the type of an object is not checked at runtime. Instead the type checker looks for the presence of a given method or attribute.
- **The idea is that it doesn't actually matter what type my data is - just if i can do with it what i want.**
- Duck typing implements polymorphism in Python

# 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 imitates a quack"
    def fly(self):
        return "The parrot is flying"

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

## Duck typing: example (2)

```
v = [Duck(), Parrot(), Man()]  
  
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: example (3)

```
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:

```
Quaaack  
The parrot imitates a quack  
The man imitates a quack too
```

# Duck typing vs Static typing (1)

```
class Car:
    def __init__(self, engine):
        self.engine = engine
    def run(self):
        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 vs Static typing (2)

```
class ElectricEngine:
    def turn_on(self):
        return "zzzz"

class EngineV8:
    def turn_on(self):
        return "brooom"

electric = ElectricEngine()
fueled = EngineV8()

car1 = Car(fueled)
car2 = Car(electric)
print(car1.run())
print(car2.run())

zzzz
brooom
```



# Duck typing vs Static typing (3)

In statically-typed language, like Java, if we want to pass different objects to the Car constructor, they has to be for example **different implementation of a common interface**:

- we had to declare explicit an interface (*IEngine for example*);
- declare its implementation (EngineV8 or ElectricEngine);
- explicitly define my Car parameter to be an implementation of IEngine.

# Duck typing vs Static typing (3)

```
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

- Due to the flexibility given by Duck typing is possible to add a function to a class, **at run-time**.
- In a **statically-typed** languages this behaviour is **not possible**: once a class is loaded by the classloader there's no way to modify the code.

# Add method to a class: example (1)

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

class Parrot(Bird):
    def quack(self):
        return "The parrot imitates a quack"
    def fly(self):
        return "The parrot is flying"

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

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

```
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
```

## Add method to a class: example (3)

```
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 only to "john" instance 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
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

Duck typing is a handy way to have a **polymorphic** code without relying on abstract classes/interfaces:

- a statement calling a method on an object does not rely on the declared type of the object (at compile time)
- The object simply must supply an implementation of the method called, when called, at run-time.
- It gives flexibility to the language.
- Doesn't guarantee the absence of type error during run-time.

