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:

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

Semantic: Python vs Haskell

In Python we never get *true* beacause it forces 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:

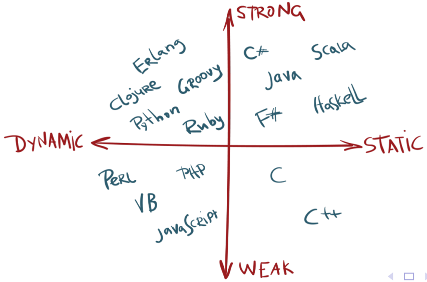
```
elem 2 [2, 4, noreturn 5]
```

Type checking

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**
    - Every sub expression is well typed
    -

Python's type checking

1. Python is **dynamic**:
   - objects have a type but it is determined at runtime;
   - variables are not explicitly typed;
   - an assignement binds a name to an object and the object could be of any type;
2. Python is also **strongly typed**.



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 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'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.

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**.

```python
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)

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

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

Object oriented (3)

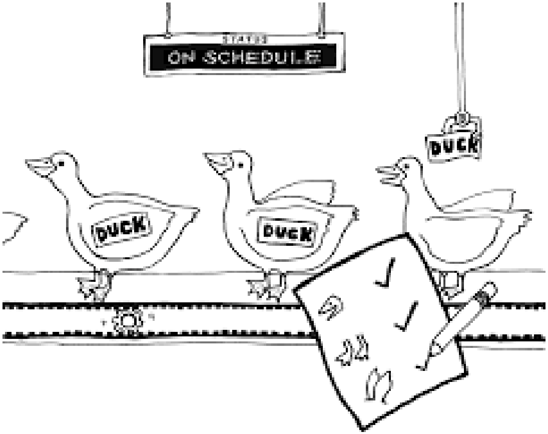
In Python **is not possible to define multiple constructor** for a class, still is possible to define a default value if one is not passed.

```
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.
- 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.
- **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)

```python
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())
```

Duck typing: example (2)

```
        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:

```
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: 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".**

Duck typing: dependency injection (2)

In a language **without** duck typing 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, due to the duck typing, there is the chance to add methods to the classes.

- **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

```python
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())
```

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

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

```
            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
TypeError: fly() takes exactly 1 argument (0 given)
```

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

To bound the method properly 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 istance "jack", proving that we added the method fly only to one istance of Man.

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