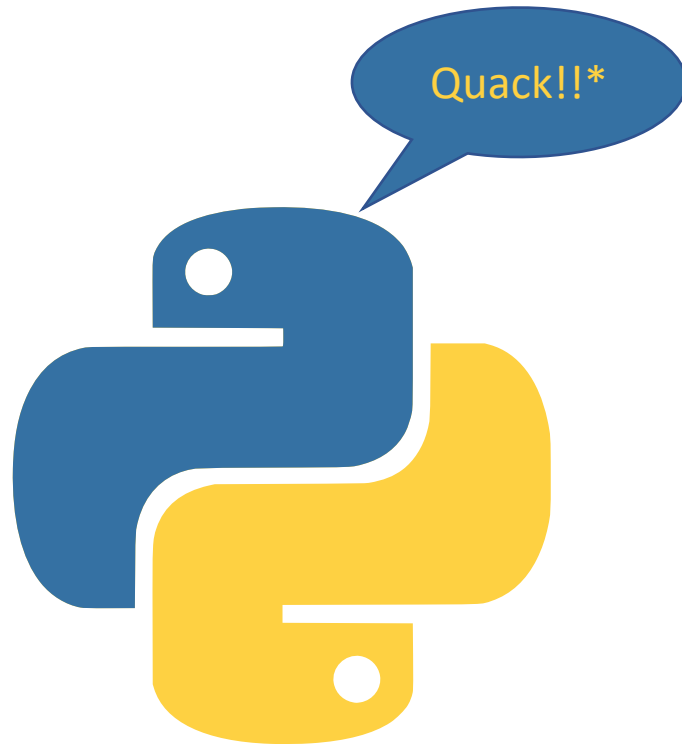


# Object Oriented Programming in Python



\* That's actually a spoiler, you have been alerted.

# Outline

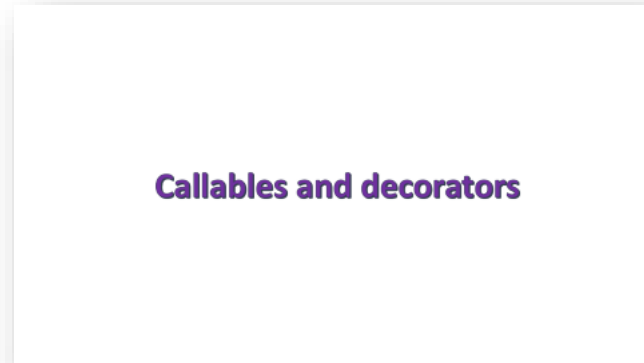
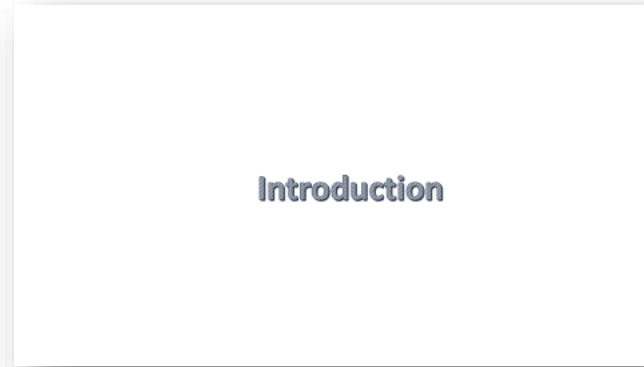
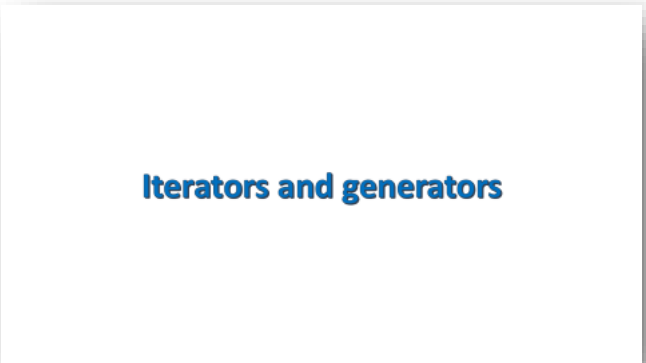
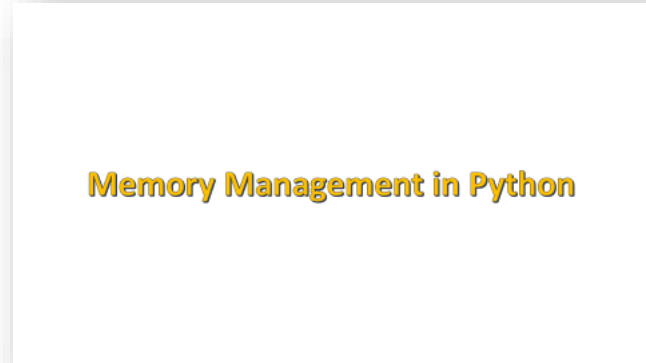
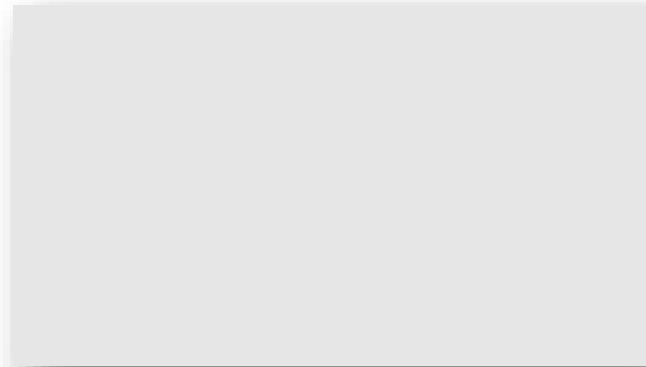
## Warm-Up



## Work-out



## Advanced Topics



# Introduction

# Is OOP funny ?

Sure! Here's a funny 5-line explanation of OOP inheritance:

DadClass: has a car, a wallet, and a bad sense of humor.

ChildClass(DadClass): gets the car, the wallet... and unfortunately, the jokes.

Overrides the jokes with memes, but still hears dad jokes at runtime.

GrandChildClass now confused: is it a Tesla or a horse and buggy?

Moral: you can inherit the goods, but debugging the baggage is on you.



1. En POO, tu crées des objets, comme des Pokémon mais pour le code.
2. Chaque objet a des pouvoirs (méthodes) et des stats (attributs).
3. Les classes sont les plans secrets pour fabriquer ces Pokémon.
4. L'héritage ? C'est quand ton nouveau Pokémon hérite des attaques de Pikachu.
5. Et l'encapsulation ? C'est cacher les bêtises pour que seul le dresseur (toi) puisse y toucher. 🧑🏻💻



Probably not

# Is OOP funny ? **Definitely not !!!**

Q: How many software developers does it take to change a light bulb?

A: None. That's a hardware problem.

There are 10 types of people: Those who understand the binary notation and those who don't...

( comment: hahahhahahhahahaha i can't stop laughing...)

# Is OOP useful ? **Definitely yes !!!**

It allows you to write :

**Shorter code**

**Versatile libraries**

**Clearer code**

**Easy to use libraries**

**Easy to use Graphical User Interfaces (GUI)**

**Easily extendable libraries**

**“Elegant” code**

# A short overview of the history of OOP

- > 1960s Firstly introduced in Simula
- > 1970s Fully implemented in Smalltalk
- > 1980s Becomes popular thanks to C++
- > 1990s Very popular languages as python and Java rely on the use of OOP
- > 2000 to present OOP is used in Javascript, C#  
New paradigms start to develop (Go, Rust, Dart ...)

Some programming languages (as Java) strongly rely on OOP

Python hello world program

```
if __name__ == '__main__':  
    print("Hello World !")
```

Java hello world program

```
public class HelloWorld {  
    public static void main(String[] args) {  
        System.out.println("Hello, World!");  
    }  
}
```

# What is OOP ?

It is a programming strategy/paradigm using entities called “objects”.

Despite the fact that some examples found in real life are not  
objects in python can be really useful !!



Translation:

Glass breaking hammer  
Break the glass to take the hammer

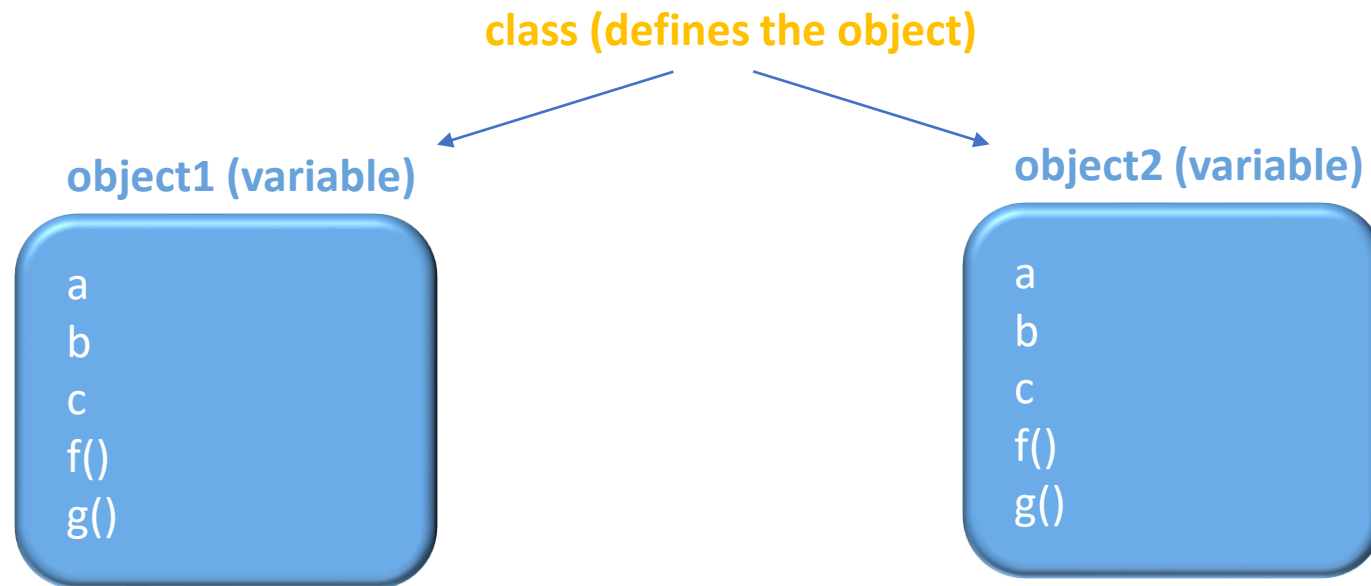


# Memory Management in Python

# What is the definition of an object in OOP? **Code**

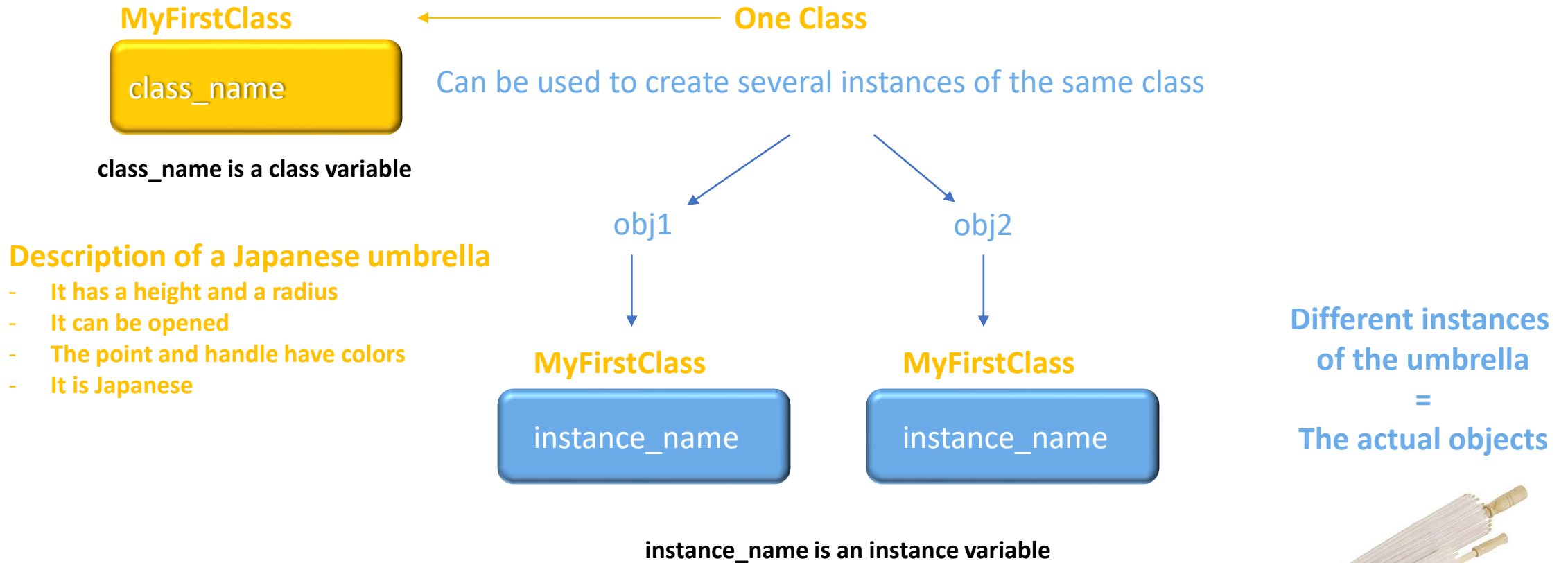
In python an object is an abstract entity that contains variables and methods (functions).

The fact that it contains “its own things (variables and functions)” is called **“encapsulation”**

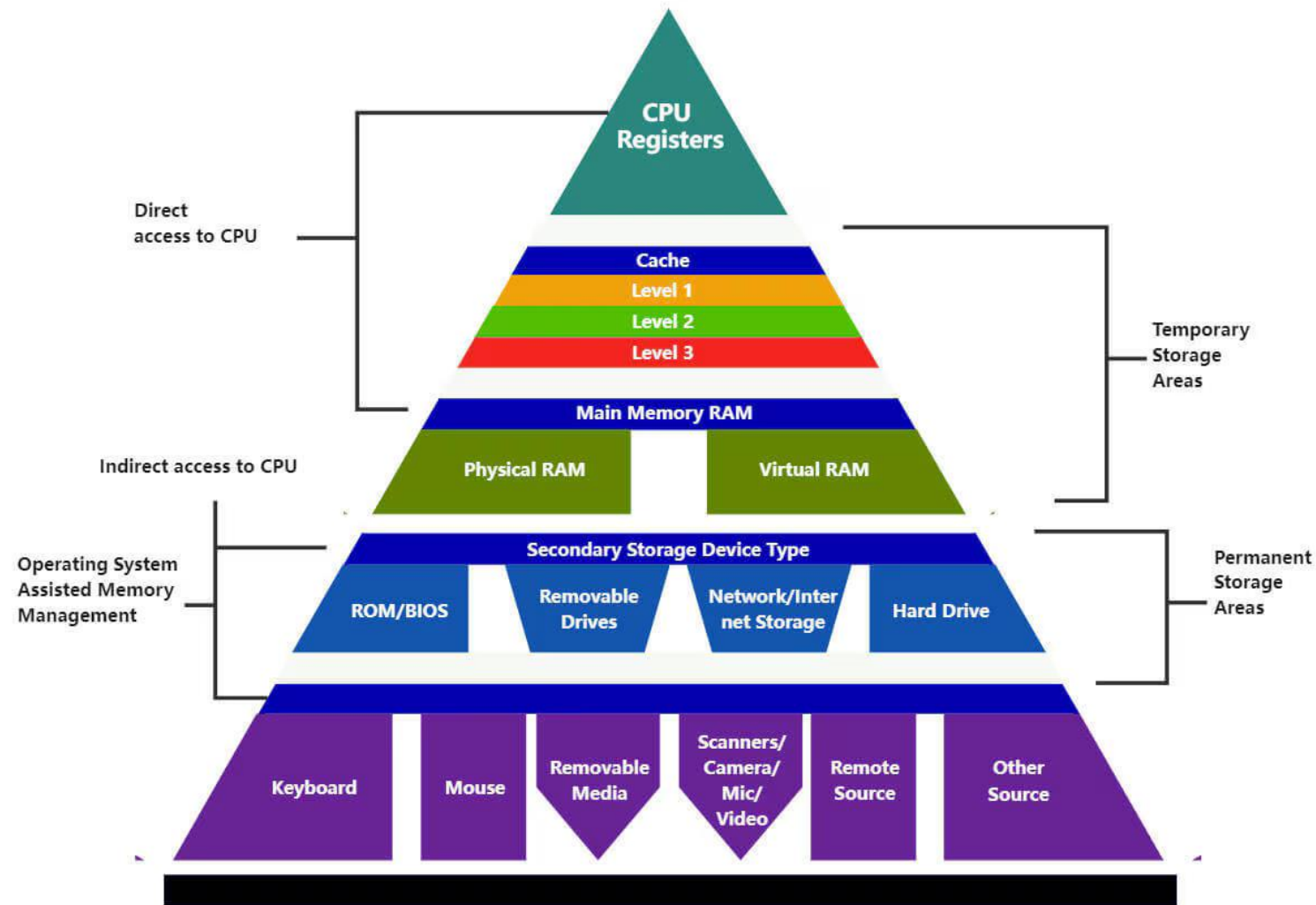


**In python everything is an object including literals !!**  
(use dir() or type() if you do not believe me)

# Objects: classes, instances and variables **Code**



# (Very) basic memory management



Memory management in modern CPUs is complex: I will use a simplified model !

# Basic memory management



Random Access Memory (RAM)

Heap memory

Used for objects and global variables

CPU Unit

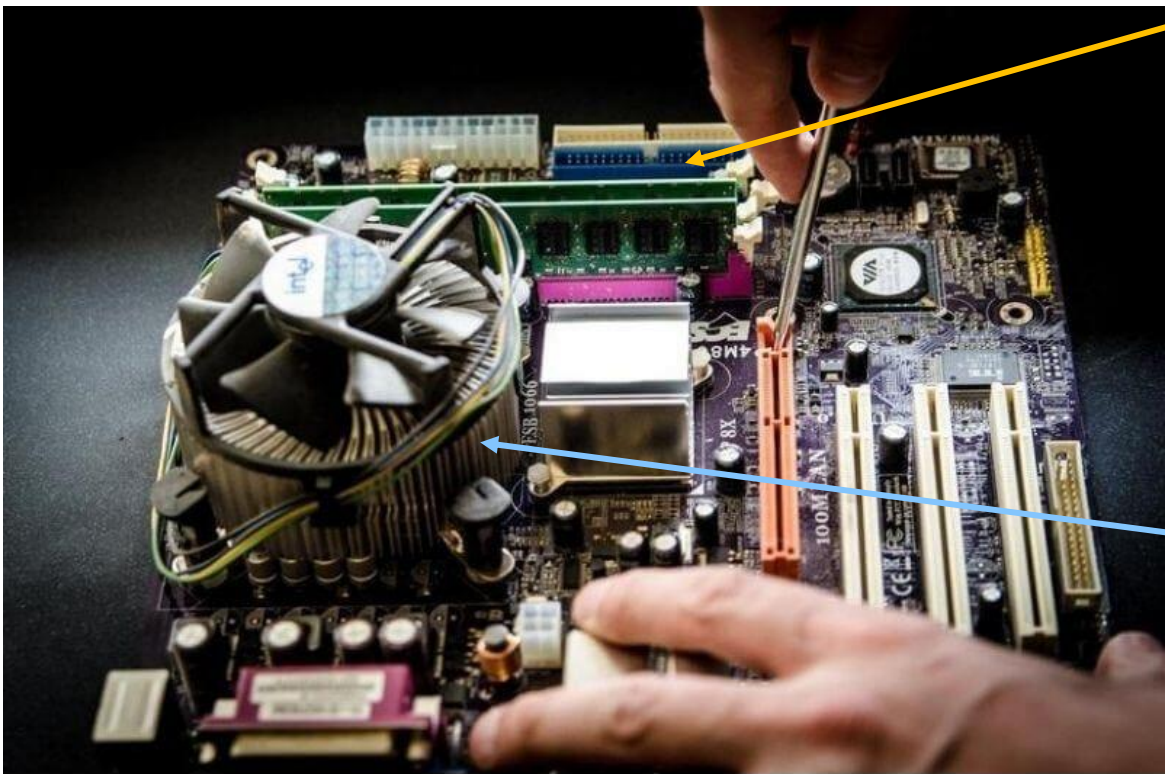
Stack memory



Microprocessor

Used for local variables  
and return values

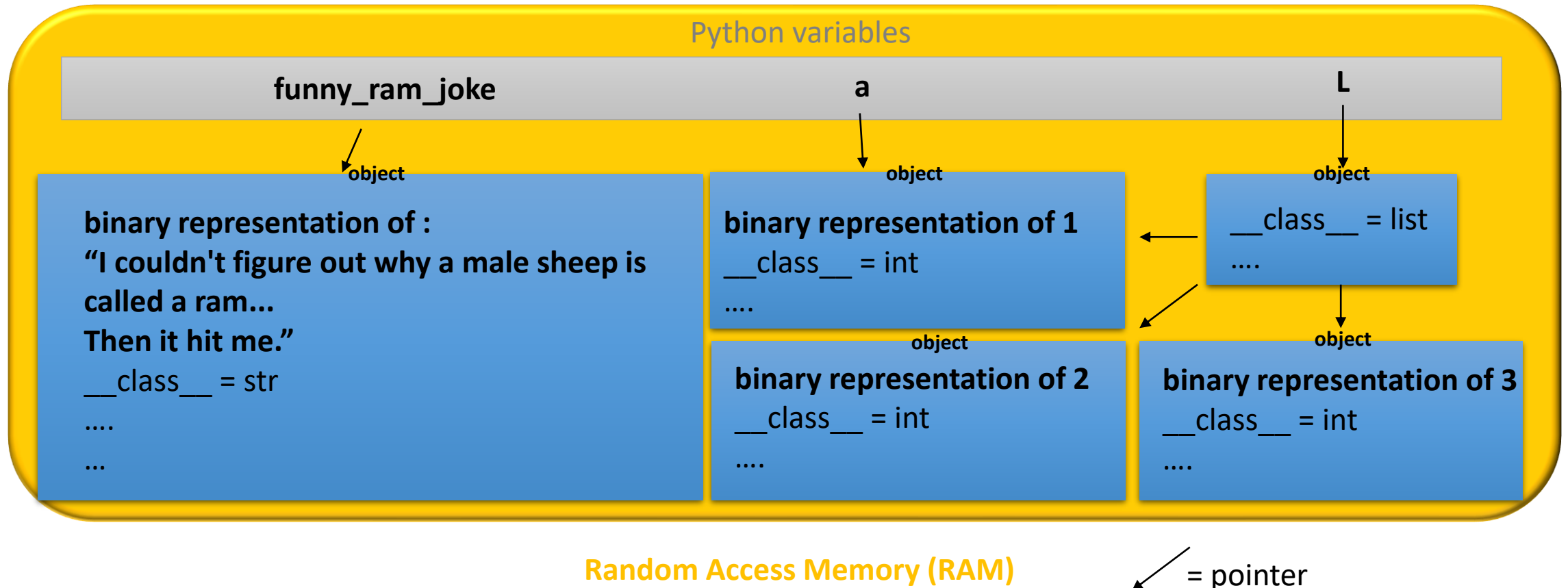
Avoid using global variables in your programs !!!



# Objects, pointers, variables and RAM

```
funny_ram_joke = "I couldn't figure out why a male sheep is called  
a ram...  
Then it hit me."  
a = 1  
L = [1, 2, 3]
```

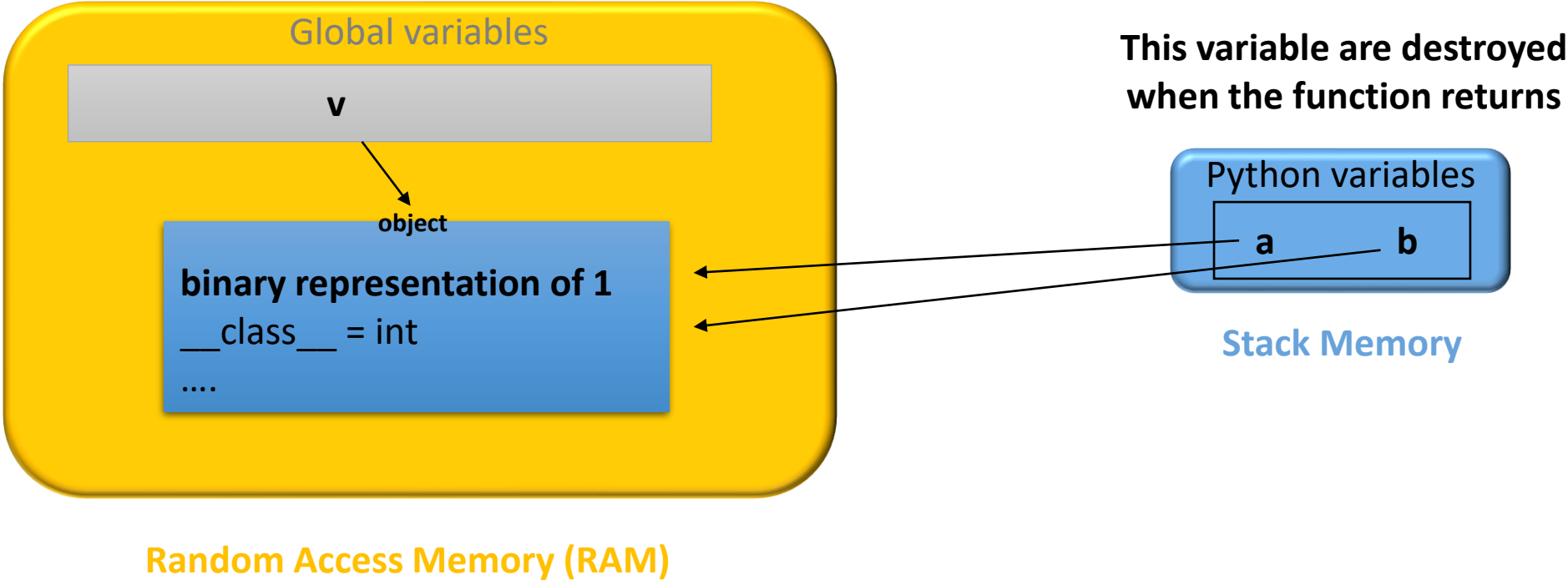
**Variables in python store a reference  
(point) to objects.  
(passing by reference paradigm)**



# Local and global variables

```
def f(a):  
    b = a  
    v = 1  
    f(v)
```

Global variables are stored in the RAM  
Local variables are stored in the stack

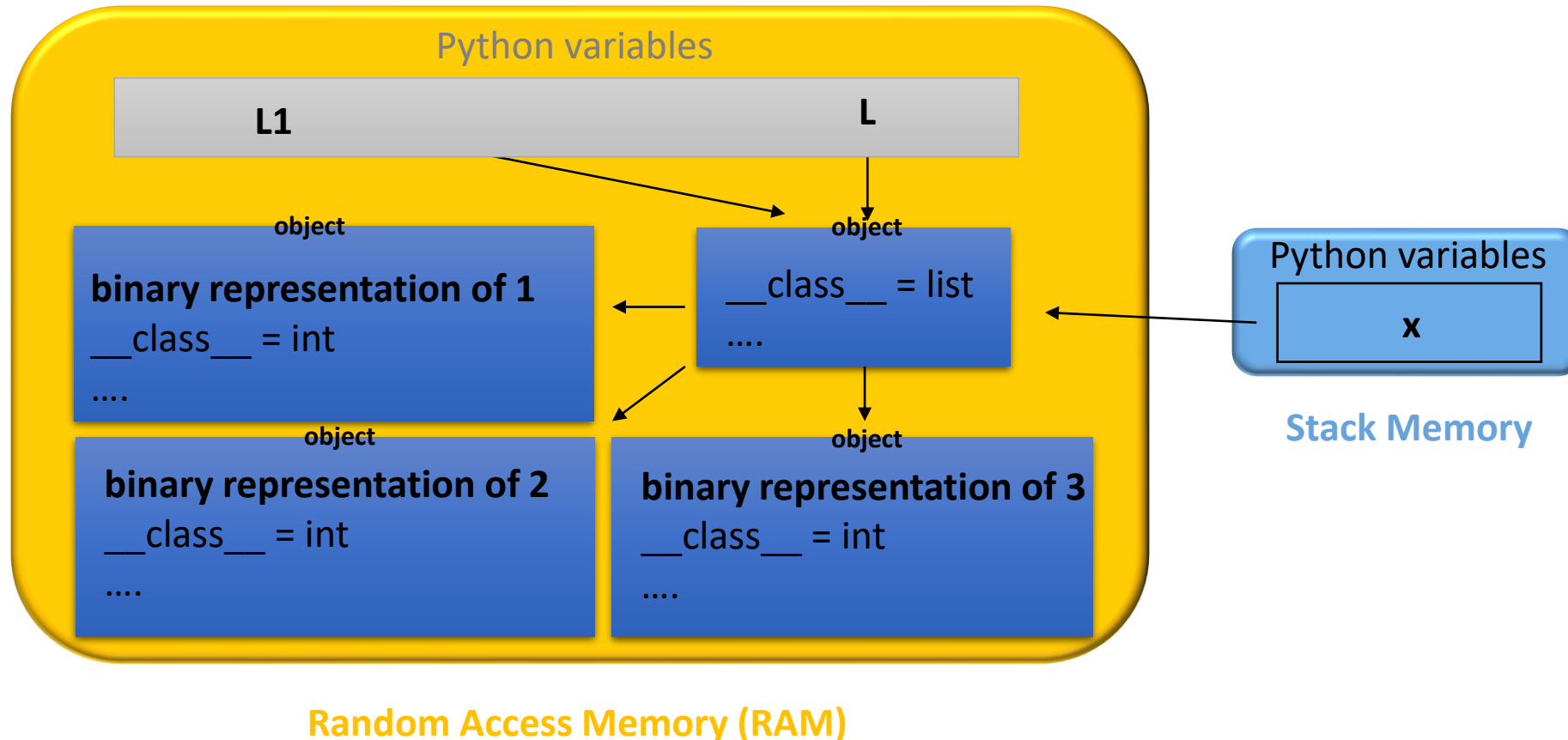


# Assignment and collateral effects **Code**

## Assignment

```
L = [1, 2, 3]
L1 = L
def f(x):
    pass
f(L)
```

**Collateral effects possible: all variables pointing the same objects are affected if you modify it using one of them**



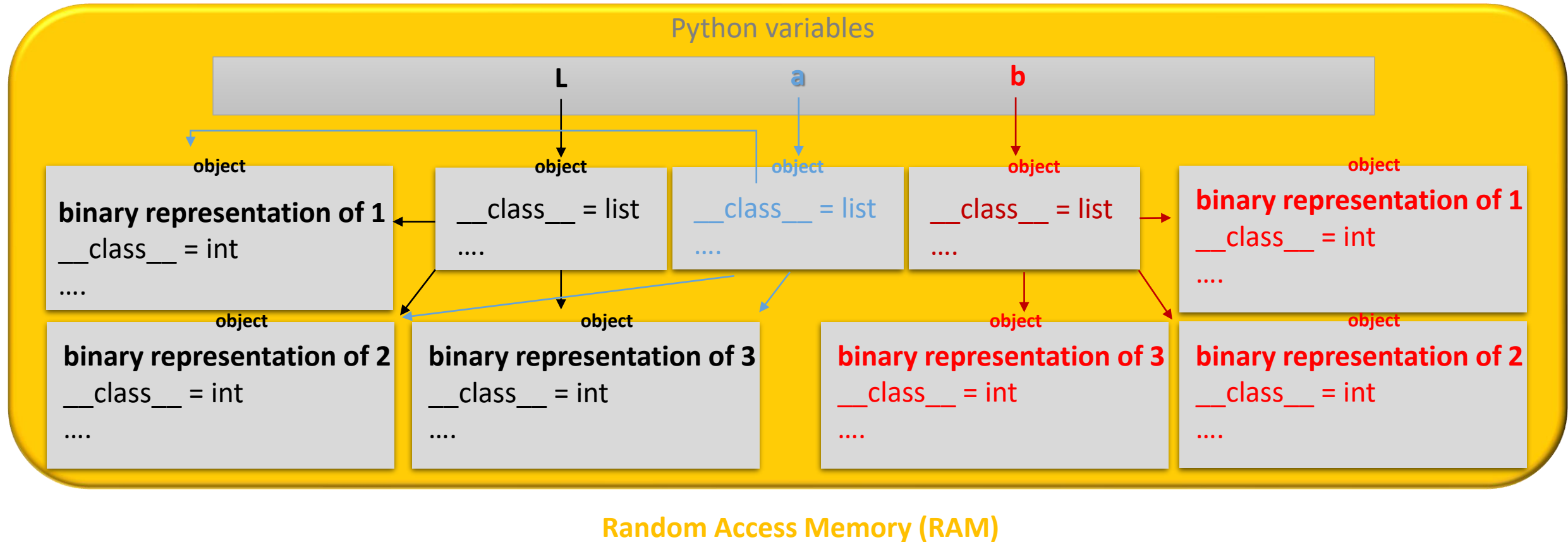


# Copy and deep copy **Code**

(shallow) copy and deep copy

```
import copy  
  
L = [1, 2, 3]  
a = copy.copy(L)  
b = copy.deepcopy(L)
```

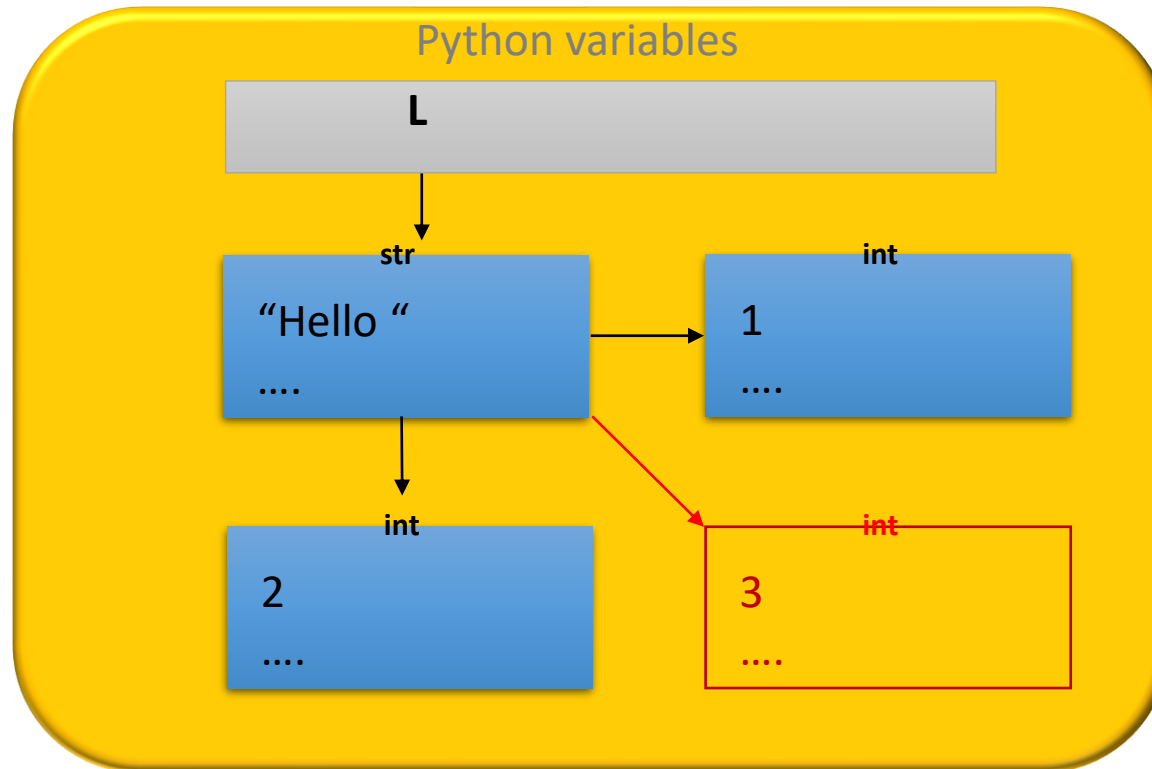
No collateral effects possible with **deep copy** (just placebo 😊) but the operation is much slower



# Mutable and immutable objects

**Mutable objects cannot be modified**

```
L = [1, 2]  
L.append(3)
```



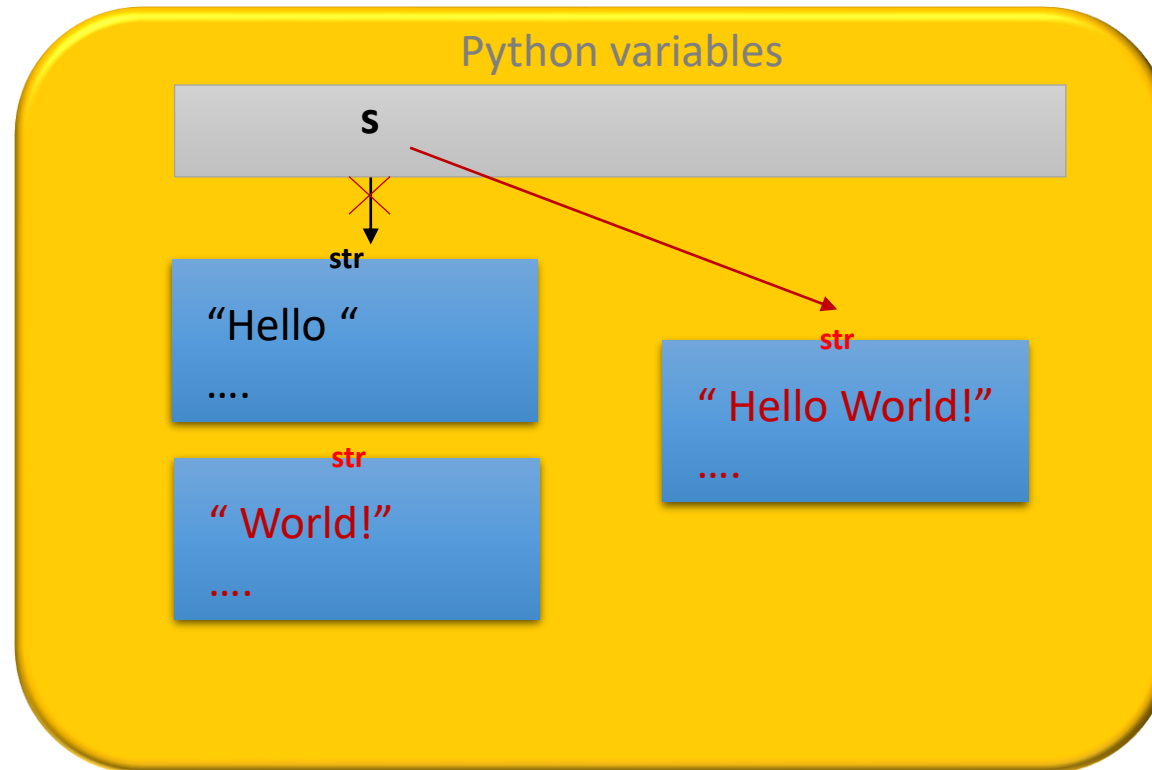
**Random Access Memory (RAM)**

**Object is modified "in place"**  
**No copy needed and that's quick !**

# Mutable and immutable objects

Immutable objects cannot be modified

```
s = "Hello"  
s = s + " World!"
```



Random Access Memory (RAM)

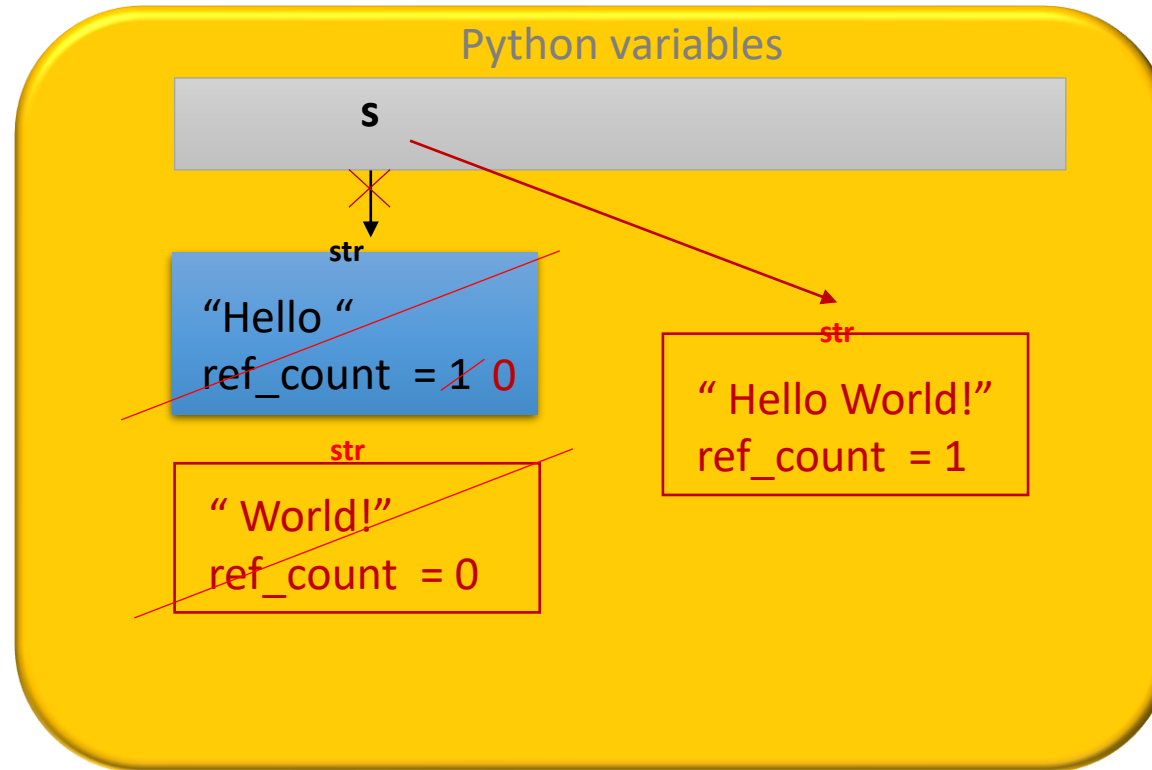
Examples: numbers, strings, tuples....  
To modify strings you need to create  
new objects !!!

What happens to all the "old" unused  
objects in memory ?

# Smart pointers and the garbage collector

Python has smart pointers

```
s = "Hello"  
s = s + " World!"
```



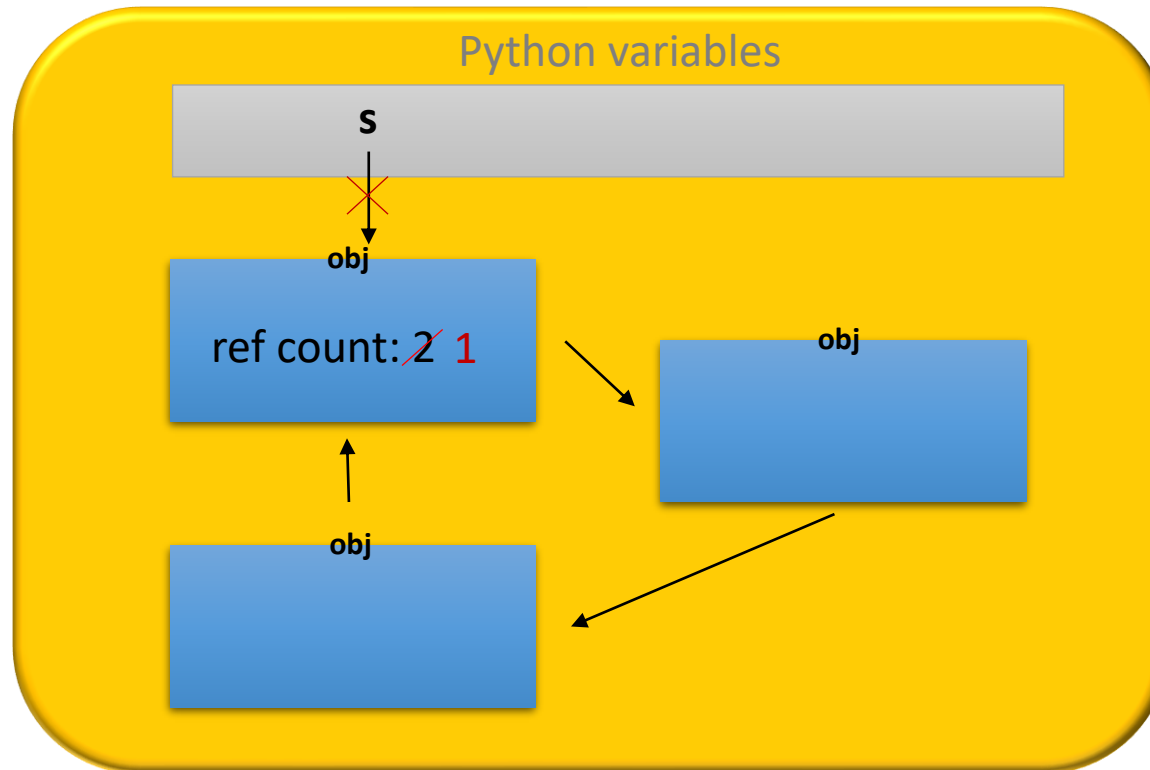
Memory is automatically freed.  
That's smart !

Random Access Memory (RAM)

# Smart pointers and the garbage collector **Code**

```
del s
```

**In some cases this does not work  
(circular references)**



Random Access Memory (RAM)

**Memory is not automatically freed**

**This is why python has a**

**GARBAGE COLLECTOR**

**That periodically checks the memory  
for the presence of such cases and  
frees them if needed**













**WHICH REDUCES PERFORMANCES**

# The smartness conservation principle

The quality of the final result is directly proportional to the amount of smartness used to produce it.

The Fazzini corollary (still top secret do not talk about it with anyone)



Pointer Smartness	Programmer Smartness	Program efficiency	Program stability
			
			
			



Votre appareil a rencontré un problème et doit être redémarré. Nous collectons simplement des informations relatives aux erreurs, puis nous allons redémarrer l'ordinateur.

0% achevés



Pour plus d'informations sur ce problème et sur les solutions possibles, consultez

<https://www.windows.com/stopcode>

Si vous contactez l'assistance, transmettez-leur ces informations :

Code d'arrêt : CRITICAL\_PROCESS\_DIED

# Section Summary

- In python everything is accessed as an object
- Objects are stored in memory and automatically deleted when not needed anymore
- Memory management has a performance cost
- You can have (unwanted) collateral effects when copying and passing variables



# Using Objects in Python



Basic concepts

Encapsulation



Inheritance



# Basic concepts

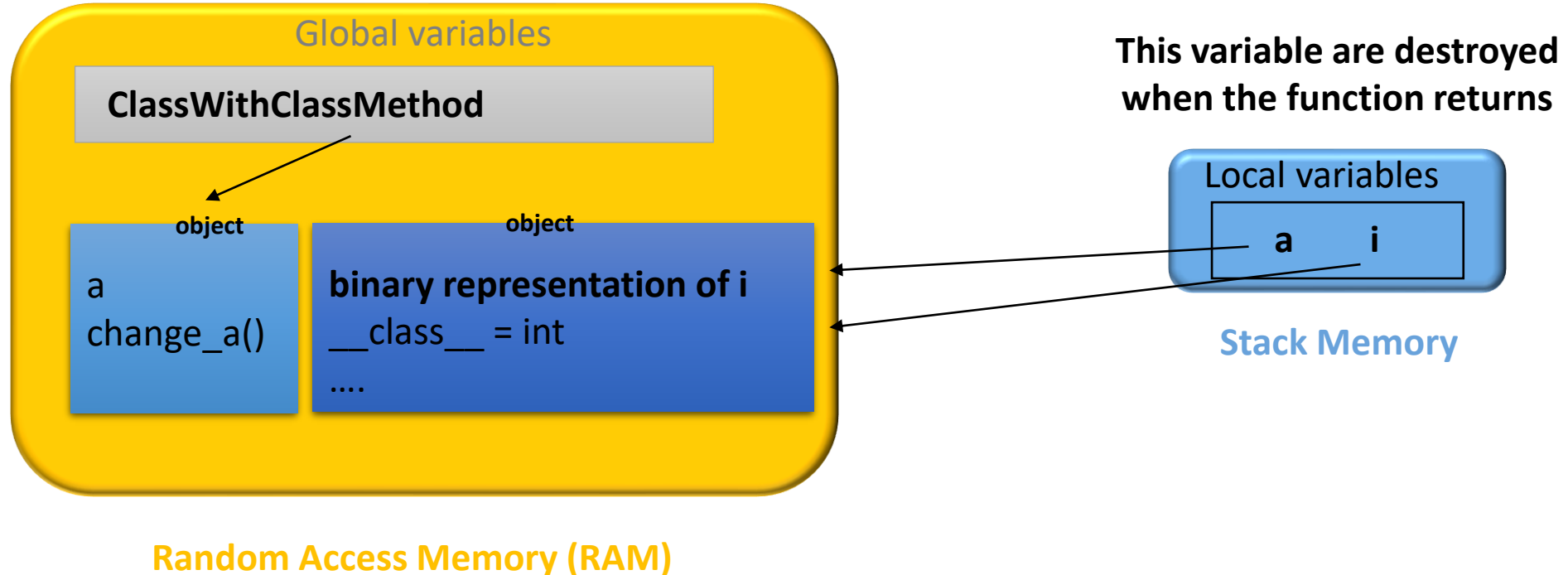
# Key Concepts

The best way (in my opinion) to manage object is

**Code**

To consider that class functions (method) do not have any special way to access class members

```
class ClassWithClassMethod:  
    a = 1  
    def change_a(i:int) -> None:  
        a = i
```



# Class methods

You can use class methods by using class objects

```
class ClassWithClassMethod:  
    a = 1  
    def change_a(i:int) -> None:  
        ClassWithClassMethod.a = i
```

To use class methods you need to use the  
`@classmethod` decorator

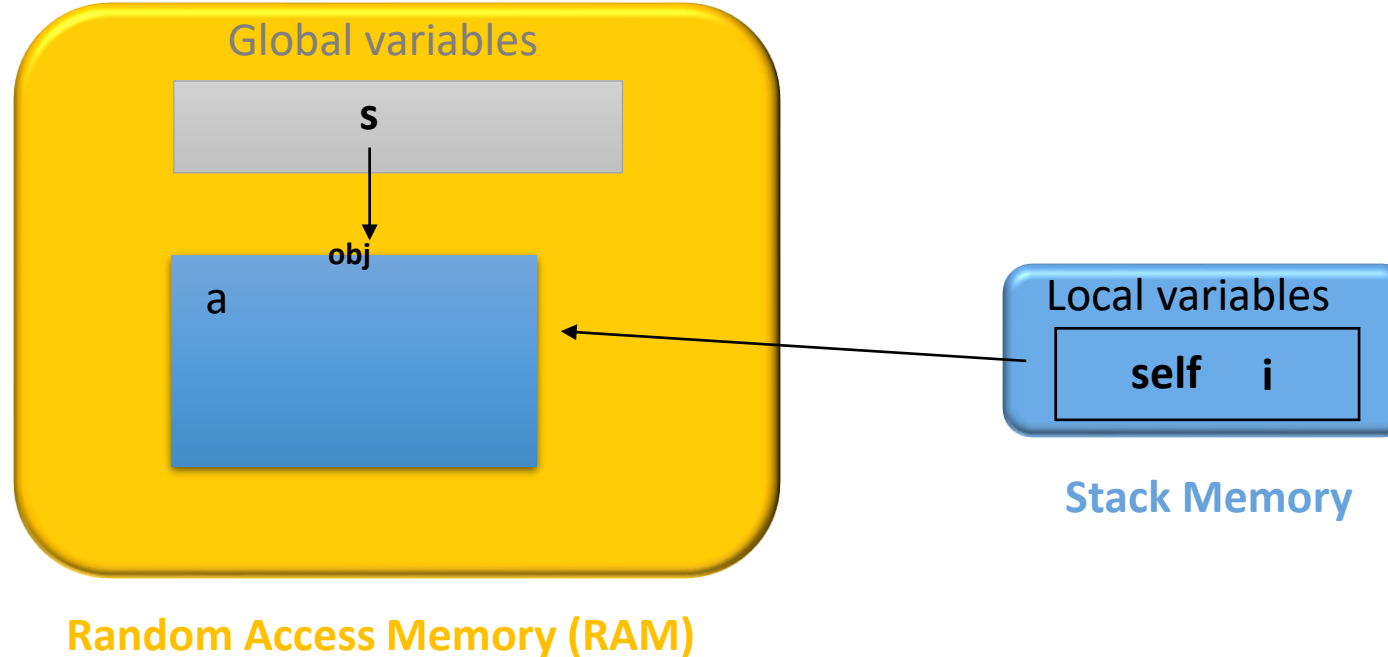
```
class ClassWithClassMethod:  
    a = 1  
    @classmethod  
    def change_a(cls, i:int) -> None:  
        cls.a = i
```

In general you will rarely need to use calls methods and  
members (you are allowed to forget this)

# Accessing Instance variables **Code**

How can I access instance variables ?

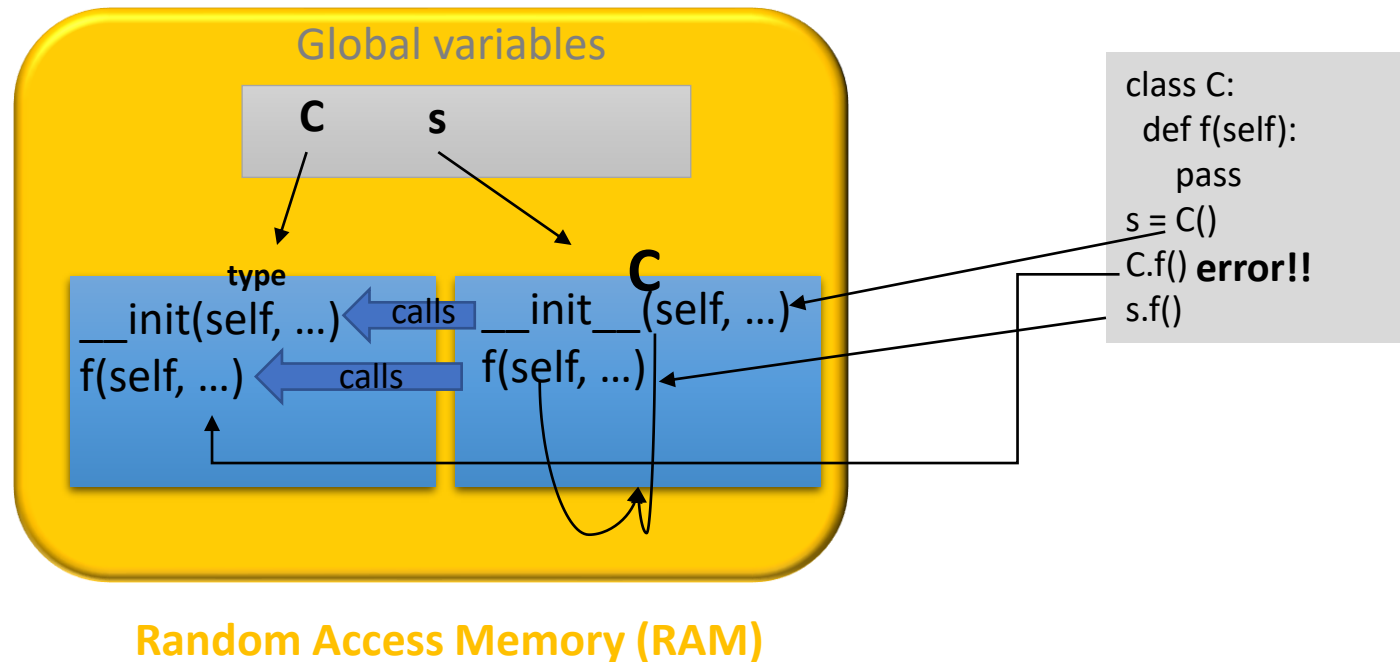
- using variables
- using **self** in class members



# Class functions and instance methods **Code**

**Class functions are automatically converted to instance methods**

- You can call a function **f** using the class
- When you call **f** from an instance of the class **self** is added as first argument
- **\_\_init\_\_** is called when an instance is created



# Encapsulation



# Encapsulation in OOP

**The basic idea of encapsulation is**

**To use objects instances as self contained pieces of code and thus to avoid using global variables.**

**The advantages of encapsulation are:**

- **Increasing performance by avoiding global variables**
- **Create code that is easy to use and to reuse**
- **Avoid errors due to conflicts of variable**

**An object is a self-contained, easy to use, hassle-free, piece of code**  
**The use of encapsulation alone makes OOP worthy !!!**



# Encapsulation in Python

**Code**

**The basic idea of encapsulation is**

**To define a “public” interface for the class user  
and a “private” implementation for the class writer**

**Public variables do not have restrictions on their name**

**Private variables and functions must begin with a `_` the class user can use them but he really should not**

**Variable beginning with `__` cannot be accessed, but the access can be forced by putting the class name in front of the variable (sometimes referred to as protected variables)**

**Any modification to the interface do not affect the user code !!!**

# Inheritance

# Inheritance

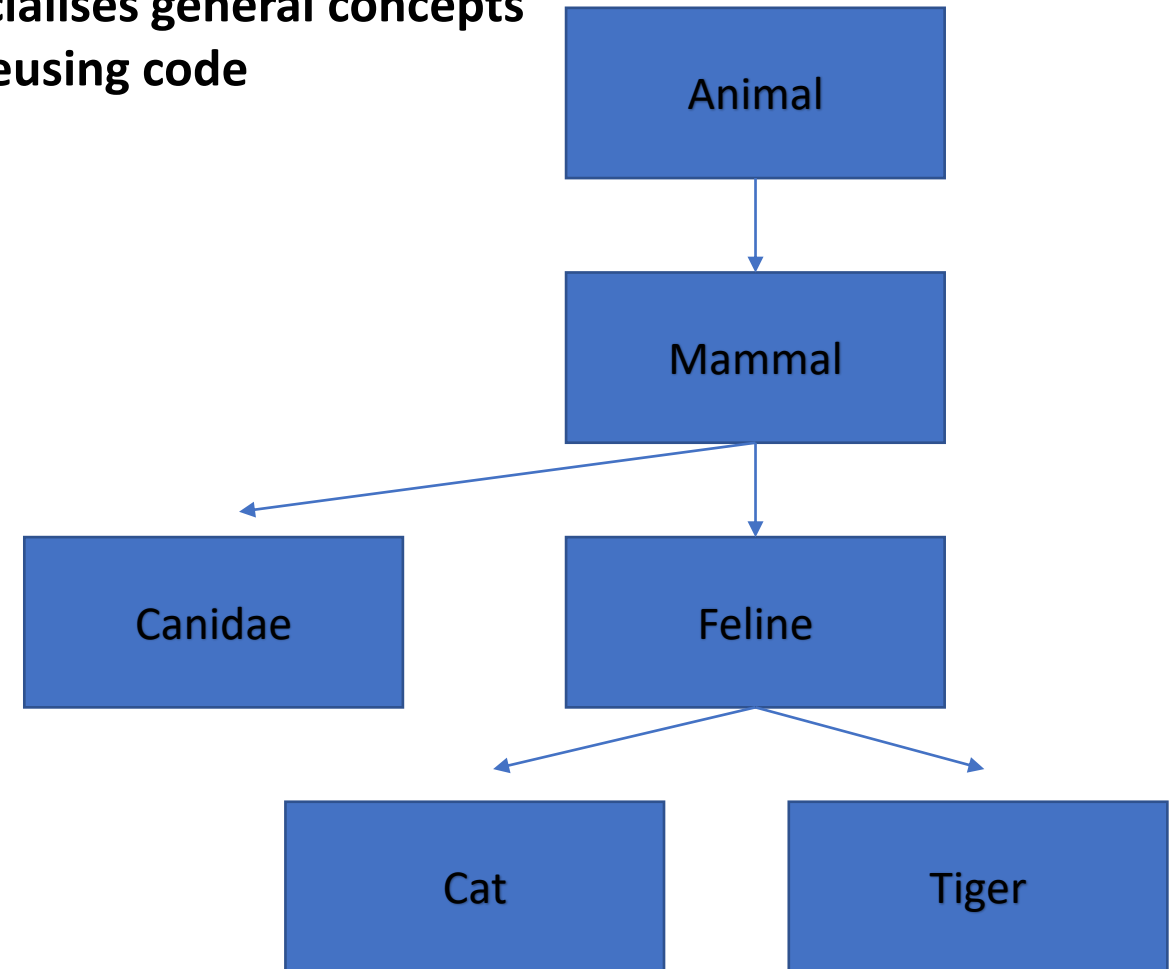
The basic idea of inheritance is

To define a hierarchy of classes that specialises general concepts to more specific tasks and reusing code

You can easily extend functionality of a class written by someone else without knowing the implementation specifics (useful in GUIs !!)

You can write general functions that works on classes with belonging to the same category (with common ancestors) ( a function Woking on all mammals).

This is called **POLYMORPHISM**.



# Polymorphism and Python

## Simple Idea (duck typing)

**If it quacks it is a duck !!!**

```
def test_duck(f):  
    f.quack
```

No error: it is a duck

Error: it is not

**But it can generate a lot of errors at runtime**

**If the programmer is not aware !!!**

# Inheritance

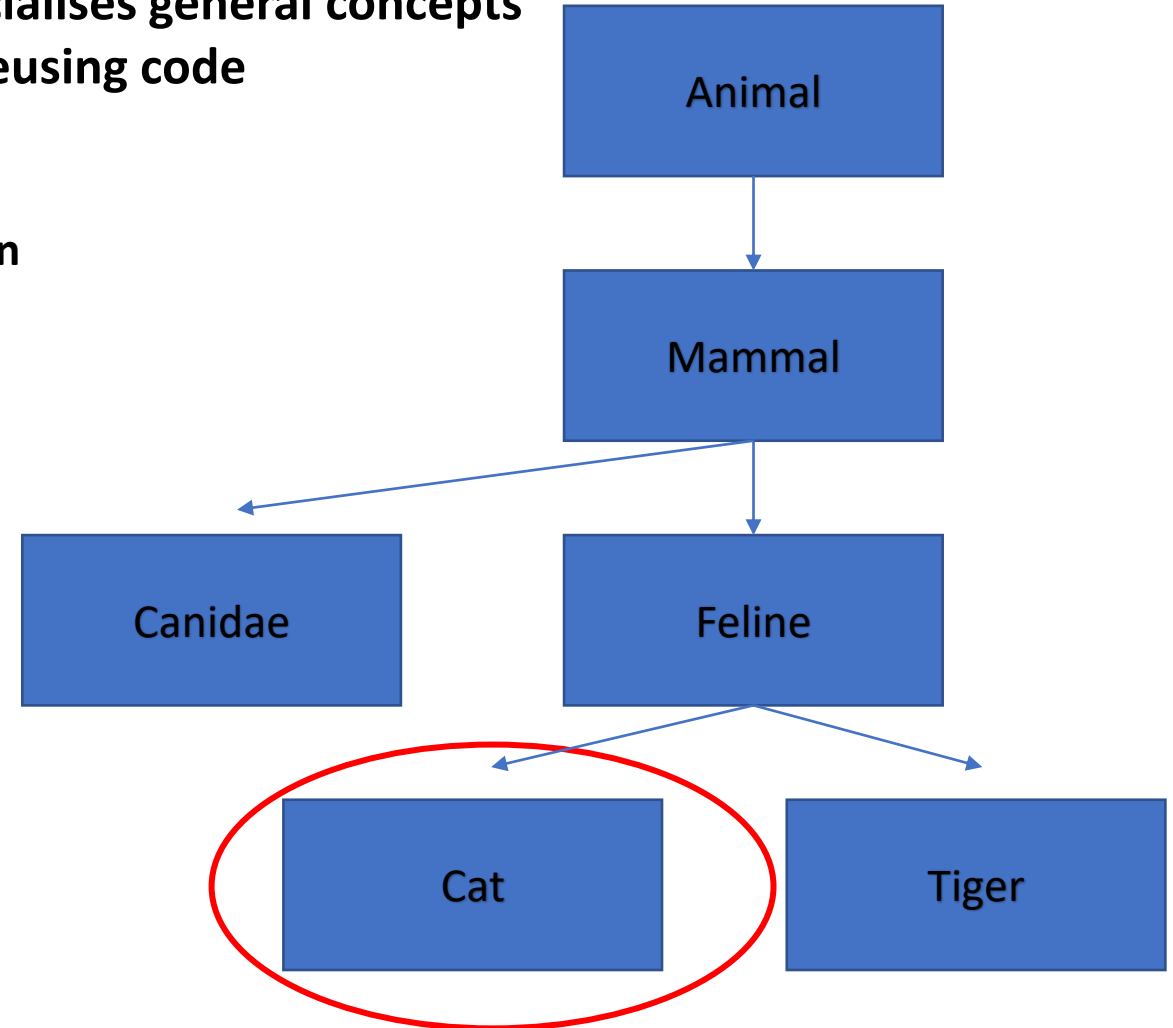
**The basic idea of inheritance is**

**To define a hierarchy of classes that specialises general concepts to more specific tasks and reusing code**

**The class can access all variables of the superclasses as its own with the exception of variable beginning with `__`**

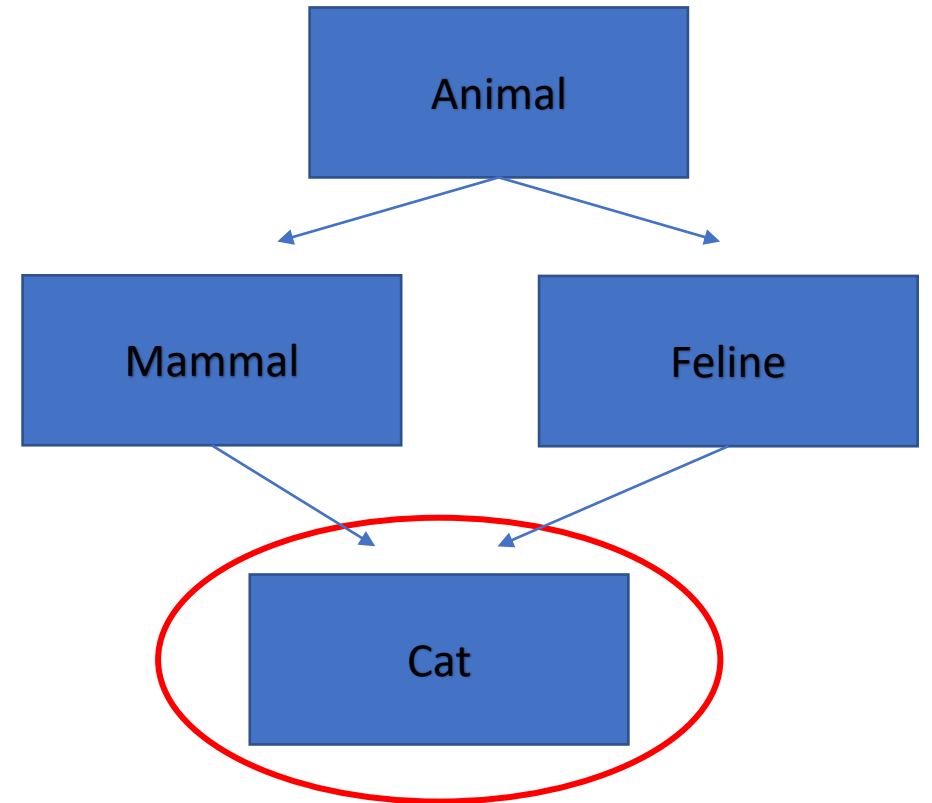
**Any function of a superclass can be called:**

- **By using the class name:**  
ex. `Feline.jump()`, `Feline.__init__()`
- **By using `super()`**  
ex. `super().jump()`, `super().__init__()`



# The diamond problem

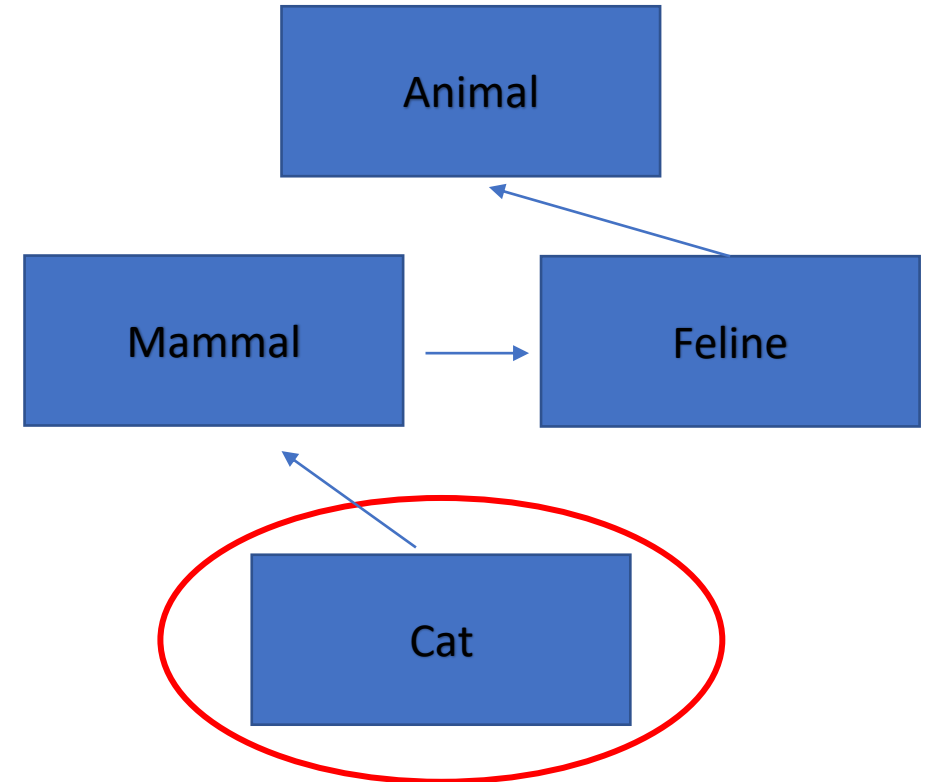
**But in this case who is super referring to ?**



# MRO (method resolution order) **Code**

The general case is complex (recursive algorithm)  
but in 99,9% of cases:

**From left to right then from bottom to top**



# Iterators and generators



# Iterators and iterables

If you can do that

```
obj = ObjectClass()  
for i in obj:  
    print(i)
```

ObjectClass is an iterable

**An iterable is an object with a special method called `__iter__`**  
**`__iter__` returns an iterator**

**An iterator is an object with a special method called `__next__`**  
**`__next__` can do two things:**

- **Return the next value in the sequence**
- **Raise a StopIteration exception to indicate the end of the sequence**

**If StopIteration is never raised the iterator never stops !!!**

# Iterators and iterables

If you can do that

```
obj = ObjectClass()  
for i in obj:  
    print(i)
```

ObjectClass is an iterable

**An iterable is an object with a special method called `__iter__`**  
**`__iter__` returns an iterator**

**An iterator is an object with a special method called `__next__`**  
**`__next__` can do two things:**

- **Return the next value in the sequence**
- **Raise a StopIteration exception to indicate the end of the sequence**

**If StopIteration is never raised the iterator never stops !!!**

**Do not use it in for loops .**

# Lazy containers

Code

**Iterators and iterables are called lazy containers**

**Because they generate the number only when needed**

**If you do not want to use classes you can use function generators**

```
def fgen(max:int):
```

```
    i = 0
```

```
    while i < max:
```

```
        yield i
```

```
    return
```

```
for i in fgen():
```

```
    print(i)
```

The function creates an iterator

Returns i as next element in the sequence

The sequence ends when the function returns

**And for simple cases you can use generator expressions**

```
odd_gen = ( 2*n+1 for n in range (10) )
```

Generates a lazy iterator giving even numbers between 1 and 19

# Callables and decorators

# Functions are objects

Code

A function is a special kind of object, called of type `function` that has a `__call__` method.

A general object is said to be **callable** (and behaves as a function) if it has the `__call__` method.

**A function that takes a function as argument, modifies it and returns it is called a decorator**

```
def decorate(f):  
    print("I print this then I execute f")  
    f()  
  
def f():  
    print("Hello")  
  
decorated_f = decorate(f)
```

```
def decorate(f):  
    print("I print this then I execute f")  
    f()  
  
@decorate  
def f():  
    print("Hello")
```

**f and decorated\_f are both decorated**

**This code can be shortened by using decorator syntax**