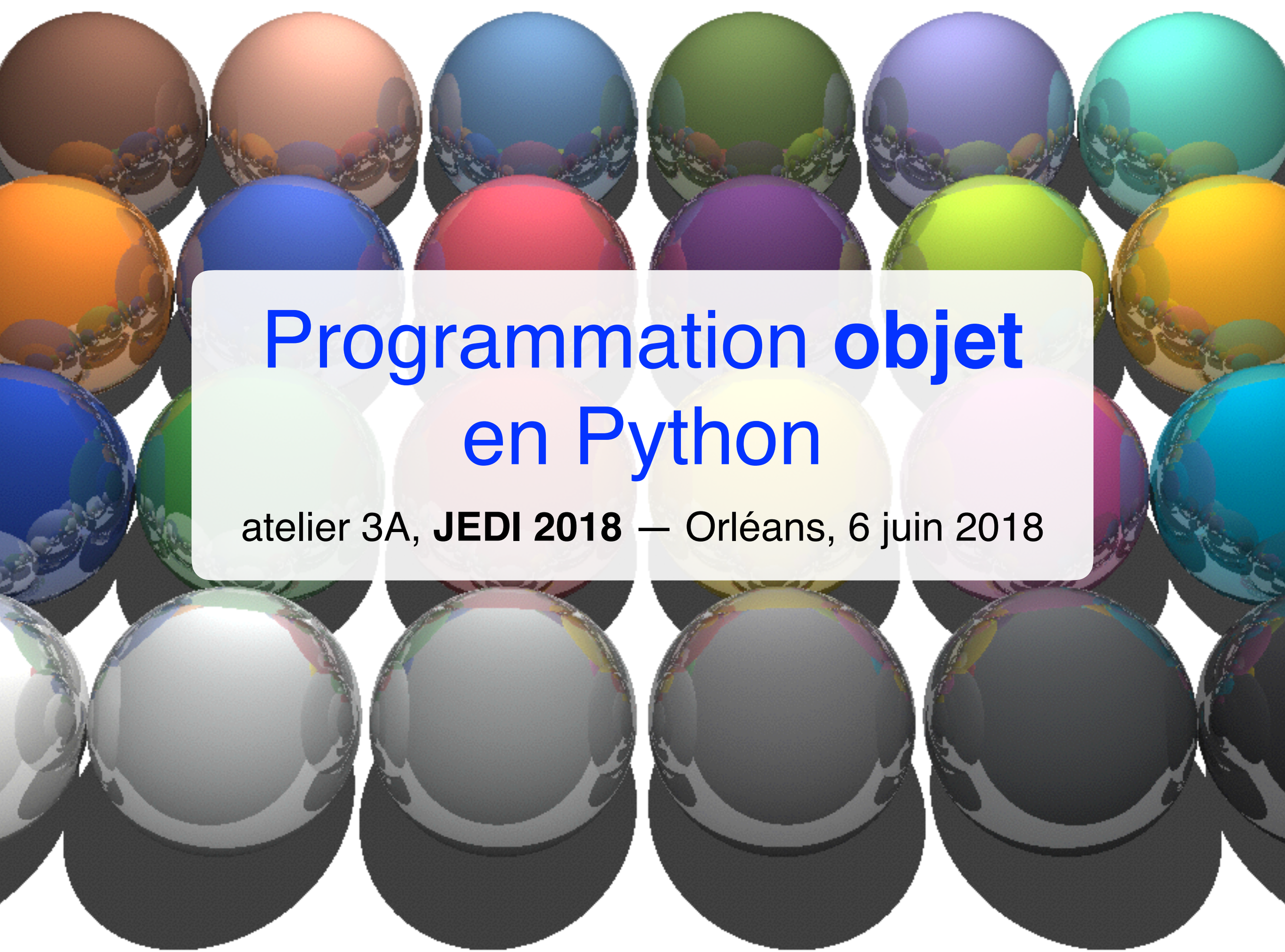


Programmation **objet** en Python

atelier 3A, **JEDI 2018** — Orléans, 6 juin 2018

Qui sommes-nous ?

Trois intervenants sur les ateliers **3A**/**3B** :

- ➡ Martin Delacourt, MdC section 27
- ➡ Noël Gillet, ATER section 27
- ➡ Nicolas Ollinger, PU section 27

Nous enseignons l'informatique à l'université d'Orléans, des réseaux à la calculabilité.

Contact : **prénom.nom@univ-orleans.fr**

Thématique 2018

Dépouillement du sondage 2016 sur les thèmes que vous souhaiteriez voir traités lors de ces journées :

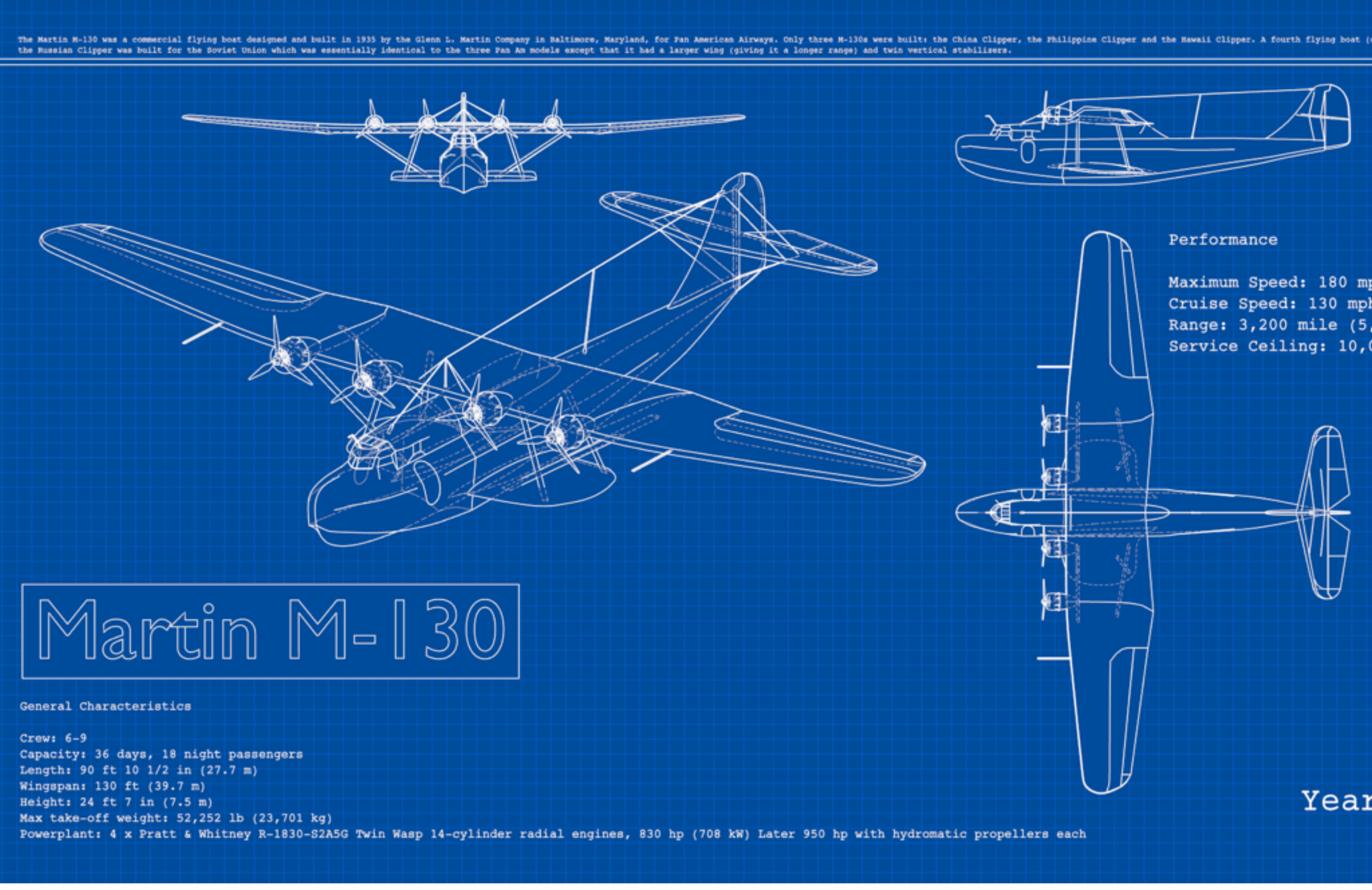
1. Bases de données
1. **Programmation objet**
1. Arduino / Raspberry Pi
1. HTML / CSS / Javascript
5. (...)

Au programme

« Qu'est-ce que la programmation orientée objet ? Comment l'utiliser en Python ? »

1. **un peu de théorie**
2. **en pratique en Python**
3. **introduction à l'atelier 3B**

Les problématiques de génie logiciel associées et la conception orientée objet ne seront pas abordées.

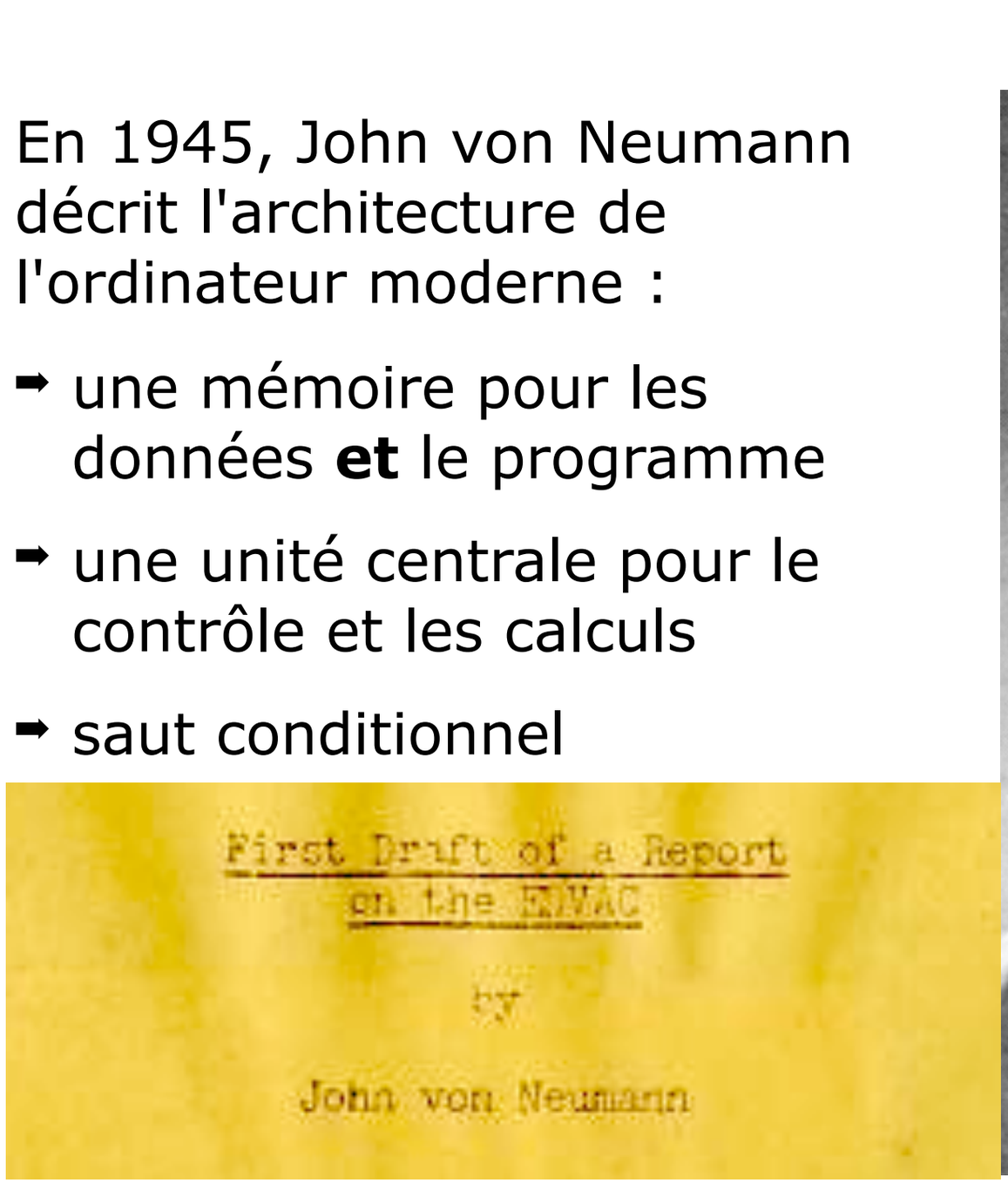


1. Programmation orientée objet

Dans les épisodes précédents...

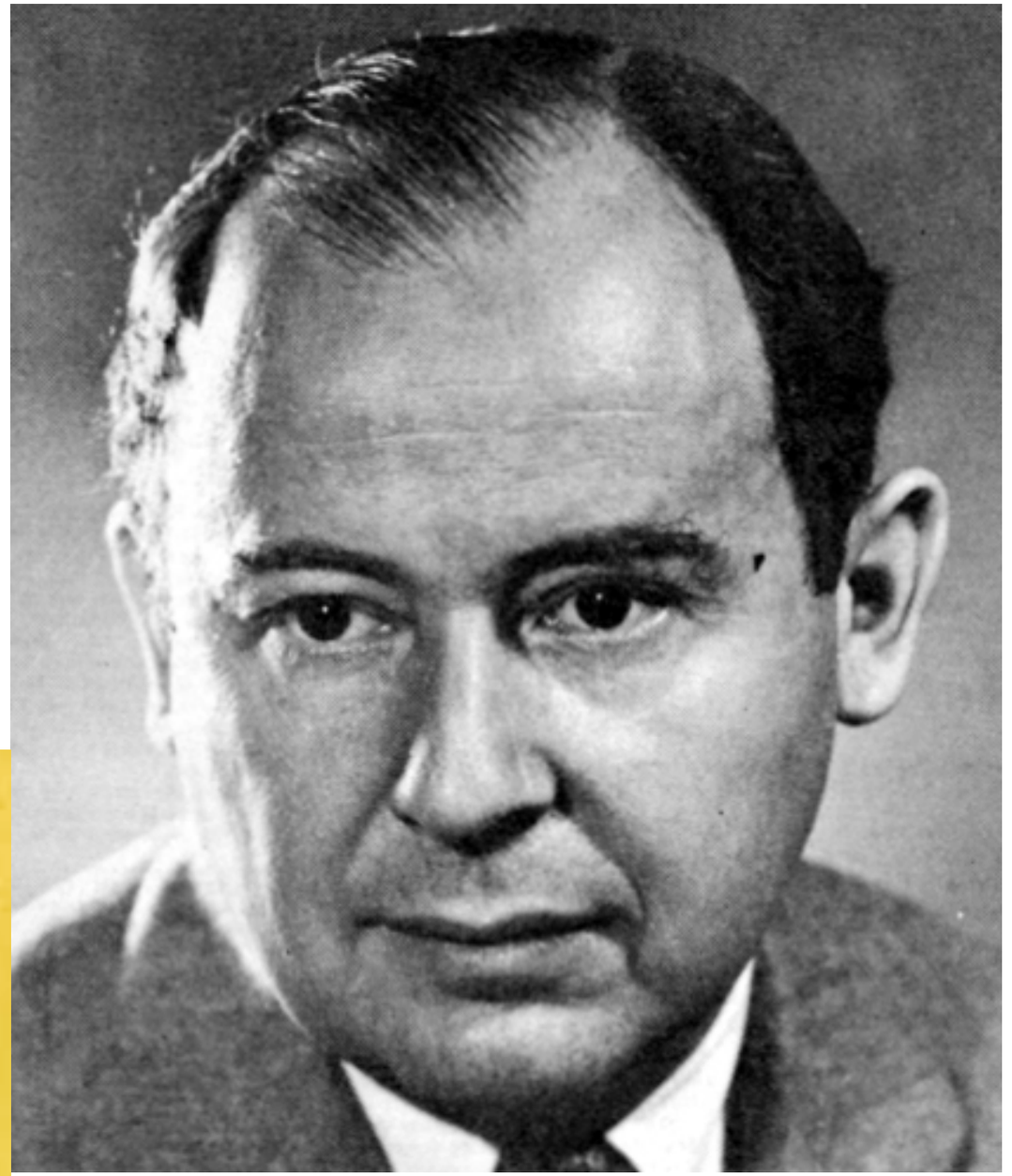
En 1945, John von Neumann décrit l'architecture de l'ordinateur moderne :

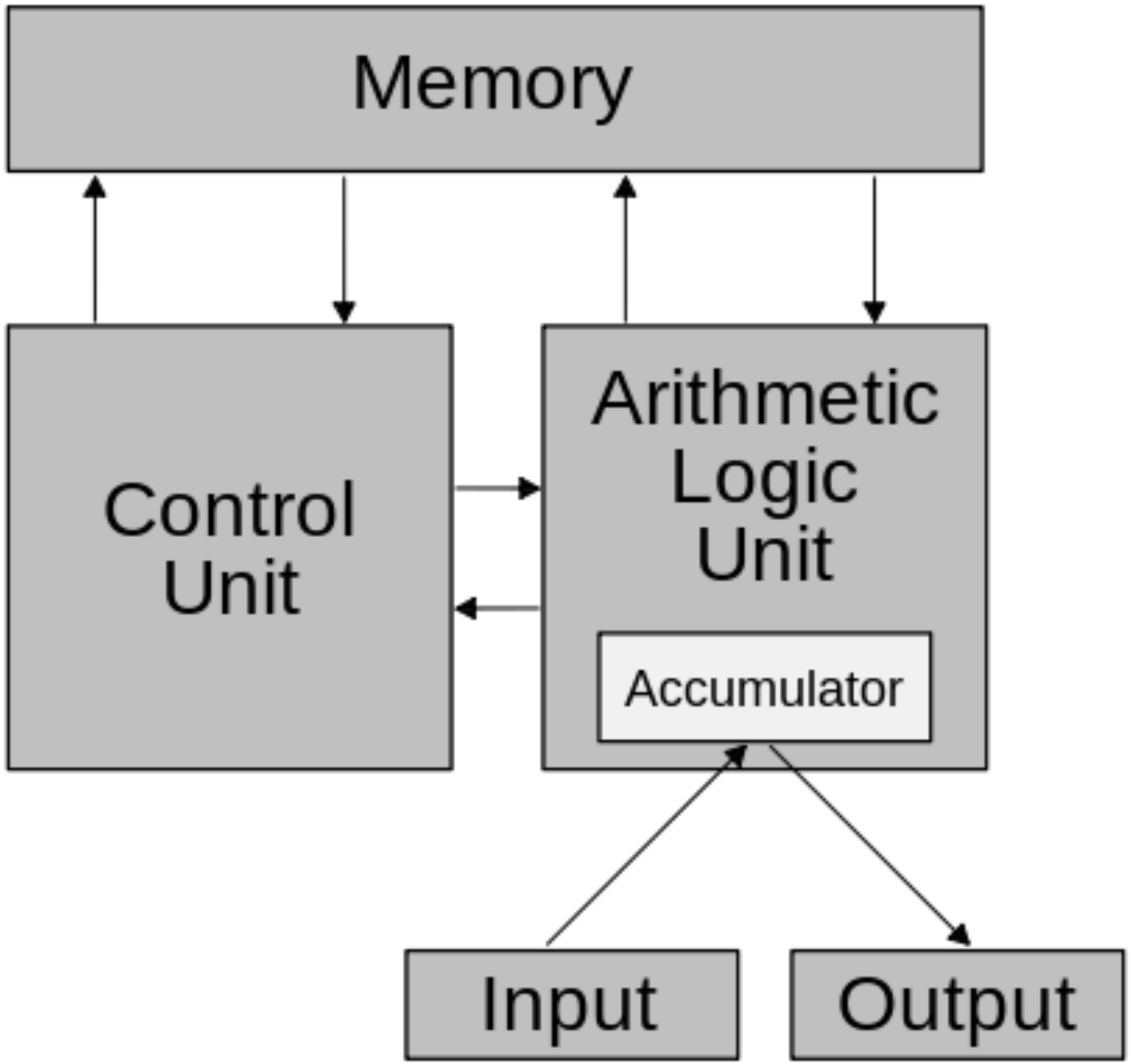
- une mémoire pour les données **et** le programme
- une unité centrale pour le contrôle et les calculs
- saut conditionnel


First Draft of a Report
on the EDVAC

by

John von Neumann




Memory
Control Unit
Arithmetic Logic Unit
Accumulator
Input
Output

Programmation impérative en Python

```
def pgcd(a,b):
    while True:
        r = a % b
        if r == 0:
            break
        a, b = b, r
    return b
```

Paradigmes de programmation

Programmation impérative (Algol, C, Pascal)

un programme décrit les instructions à exécuter pour modifier l'état du programme et atteindre le résultat.

Programmation déclarative (Prolog, SQL)

un programme décrit ce que l'on a et ce que l'on cherche à obtenir, sans expliciter comment atteindre ce résultat. Pas d'effet de bord.

Programmation fonctionnelle (LISP, OCaml, Haskell)

un programme décrit des fonctions et expressions sans effet de bord qui permettent de calculer le résultat.

Programmation orientée objet (Smalltalk, Java, C++)

un programme décrit les interactions de briques logicielles appelées objets. Le résultat est obtenu par l'interaction entre ces objets. Contrôle des effets de bord.

Programmation orientée objet

Introduite dans les années 60. **Simula** (1967) est le premier langage orienté objet.

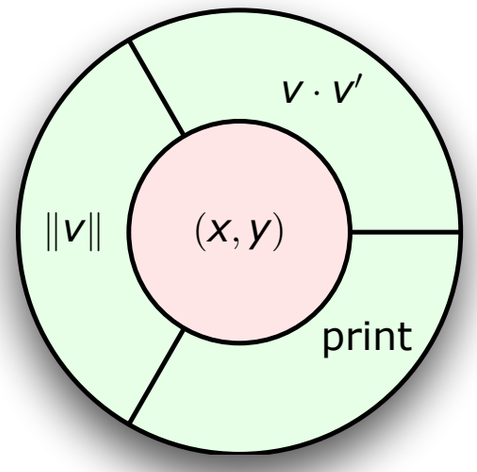
Le paradigme de programmation **dominant** depuis les années 90. Associé à des bonnes pratiques de conception et de génie logiciel pour produire du code facile à développer, maintenir, réutiliser, *etc*.

KIS (keep it simple)

DRY (don't repeat yourself)

Programmation orientée objet

Un **objet** combine des données (des *attributs*) et des sous-programmes (des *méthodes*) qui accèdent à ces données, les modifient et interagissent avec d'autres objets à travers leurs méthodes.



Vecteur 2D

Le principe d'**encapsulation** sépare la partie visible (les noms et paramètres des méthodes), **abstraite**, d'un objet de la partie cachée (les attributs et le code) qui mettent en œuvre l'objet.

Modèle objet à base de classes

Les **classes** définissent ce qui est commun à toute une classe d'objets, ce sont des types qui définissent les méthodes et la liste des attributs. Un objet est une **instance** d'une classe.

L'**héritage** permet de définir une classe comme une spécialisation d'une classe existante, on parle alors de **sous-classe**. Les méthodes peuvent être conservées, modifiées ou encore enrichies.

Le **polymorphisme** permet d'utiliser une instance d'une classe partout où sa classe parent est utilisable.

```
class Vecteur:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def length(self):
        return sqrt(self.x ** 2 + self.y ** 2)

    def dot(self, v):
        return self.x * v.x + self.y * v.y
```

2. Les objets en Python 3

Déclarer une classe en Python

Introduit par le mot-clé **`class`** :

```
class Canard:
    pass
```

Les méthodes se déclarent dans la classe avec **`def`** :

```
class Canard:
    def vole(self):
        print("flap ! flap !")
    def cancane(self):
        print("coin ! coin !")
    def nage(self):
        print("~~~~~~")
```

Instanciation et appel de méthodes

Un objet **instance** d'une classe est créé en appelant cette classe avec ses paramètres d'initialisation.

```
donald = Canard()
```

On accède aux attributs et méthodes d'un objet par la notation pointée :

```
donald.cancane()
donald.nage()
donald.cancane()
```

Initialisation et attributs

La méthode `__init__(self, ...)` est invoquée à l'instanciation de l'objet :

```
class Canard:
    def __init__(self, nom, espece):
        self.nom = nom
        self.espece = espece.capitalize()

>>> bob=Canard("bob", "colvert")
>>> bob
<__main__.Canard object at 0x10e3abf60>
>>> bob.espece
'Colvert'
```

```
class Vecteur:
    """vecteur réel 2D"""
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def length(self):
        return sqrt(self.x ** 2 + self.y ** 2)

    def dot(self, v):
        return self.x * v.x + self.y * v.y
```

Des méthodes un peu spéciales

Python définit un certain nombre de méthodes spéciales dont les noms sont préfixés et suffixés par `__` et qui permettent de surcharger les opérateurs du langage.

def `__repr__(self):` la chaîne de caractère qui représente canoniquement l'objet, retournée par **`repr`**.

def `__str__(self):` conversion de l'objet en chaîne de caractère retournée par **`str`**.

def `__add__(self, x):` résultat de l'addition de l'objet et de l'objet x (opérateur **+**)

...

```
class Vecteur:
    """vecteur réel 2D"""
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __repr__(self):
        return "Vecteur(%s,%s)" % (repr(self.x), repr(self.y))
    def __str__(self):
        return "(%s,%s)" % (str(self.x), str(self.y))
    def length(self):
        return sqrt(self.x ** 2 + self.y ** 2)
def dot(self, v):
    return self.x * v.x + self.y * v.y
def __add__(self, v):
    return Vecteur(self.x + v.x, self.y + v.y)
def __sub__(self, v):
    return Vecteur(self.x - v.x, self.y - v.y)
def __mul__(self, v):
    return self.dot(v)
```

```
>>> x=Vecteur(1,2)
>>> y=Vecteur(-3,4)
>>> z=Vecteur(2,5)
>>> x+y
Vecteur(-2,6)
>>> (x+y)*z
26
>>> str(x)
'(1,2)'
>>> repr(x)
'Vecteur(1,2)'
>>> x.length()
2.23606797749979
```

Héritage et sous-classes

Une **sous-classe** prend en argument la classe dont elle hérite. La fonction `super()` permet d'accéder aux méthodes de la classe parent.

```
class CanardAPiles(Canard):
    def __init__(self, nom, piles):
        super().__init__(nom, "rubber")
        self.piles = piles
    def status(self):
        print("batterie: %d%%" % (100*self.piles))

>>> bob=CanardAPiles("bob", 0.75)
>>> bob.status()
batterie: 75%
>>> bob.vol()
flap ! flap !
```

En Python, tout est un objet !

```
>>> type(True)
<class 'bool'>
>>> type(54)
<class 'int'>
>>> type(3.14)
<class 'float'>
>>> type("toto")
<class 'str'>
>>> type(math)
<class 'module'>
>>> def f(x,y): return x + y
...
>>> type(f)
<class 'function'>
>>> type(bob)
<class
'__main__.CanardAPiles'>
```

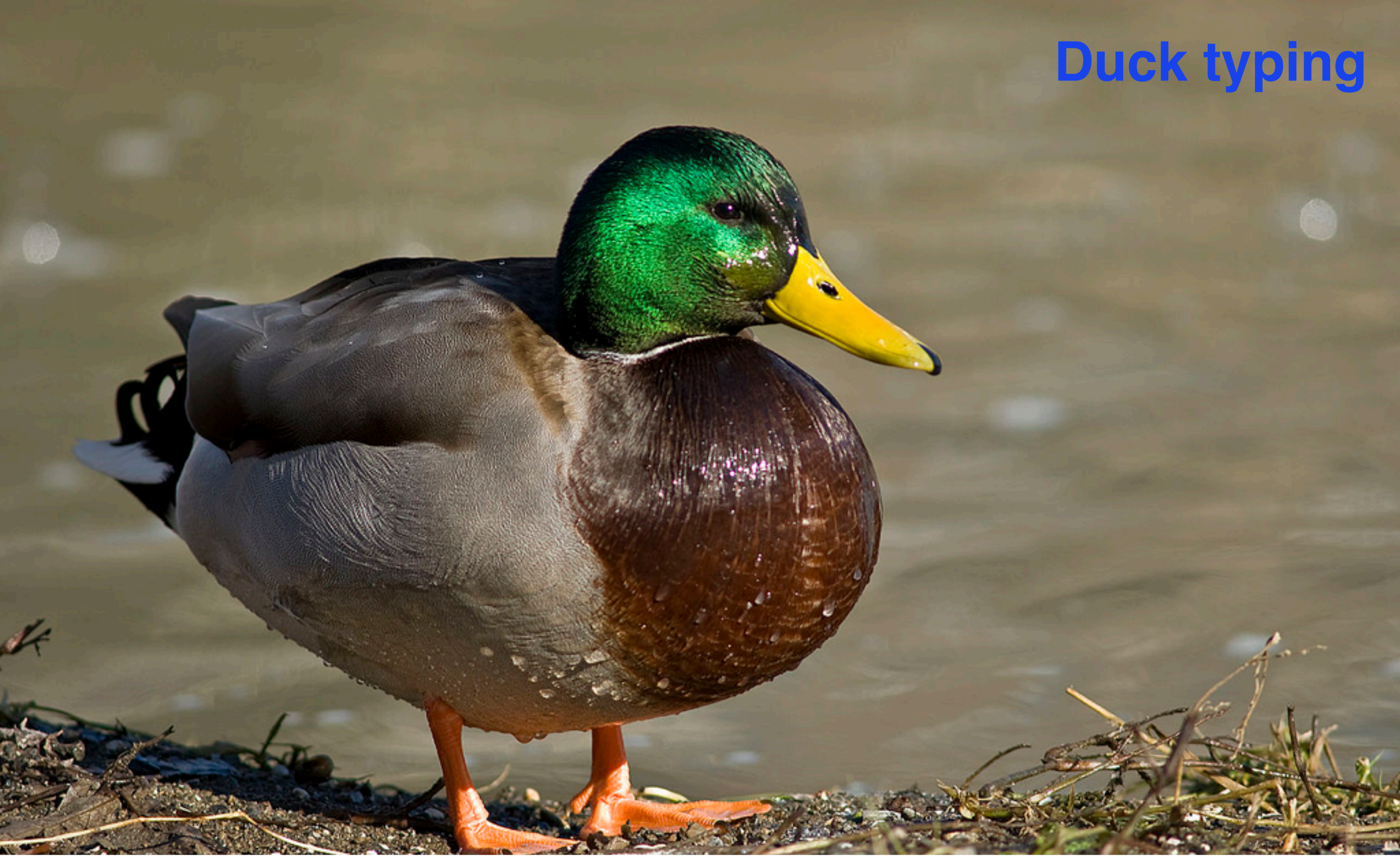
```
>>> help(float)

Help on class float in module builtins:

class float(object)
 |  float(x) -> floating point number
 |
 |  Convert a string or number to a
 |  floating point number, if possible.
 |
 |  Methods defined here:
 |
 |  __abs__(self, /)
 |      abs(self)
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __bool__(self, /)
 |      self != 0
 |
```

Polymorphisme et Duck typing

En Python il n'est pas nécessaire de dériver d'un même type de base pour pouvoir être utilisé dans le même contexte qu'un autre objet.

Du fait du **typage dynamique** il suffit de proposer les méthodes et les attributs attendus dans ce contexte.

Duck typing



« Si je vois un oiseau qui vole comme un canard, cancane comme un canard, et nage comme un canard, alors j'appelle cet oiseau un canard » (d'après J.W. Riley)

One more thing...

Certaines bibliothèques de l'atelier **3B** utilisent l'**annotation de fonctions**, introduite dans Python 3.

```
def charat(s : str, i : int) -> int:
    return ord(s[i])
```

À l'exécution, Python ignore ces annotations.

```
>>> def f(x : str, y : float) -> "toto":
...     return x+y
...
>>> f(2,5)
7
```

Pour vérifier le typage, on peut utiliser des outils externes comme **mypy**.

3. introduction à l'atelier **3B**

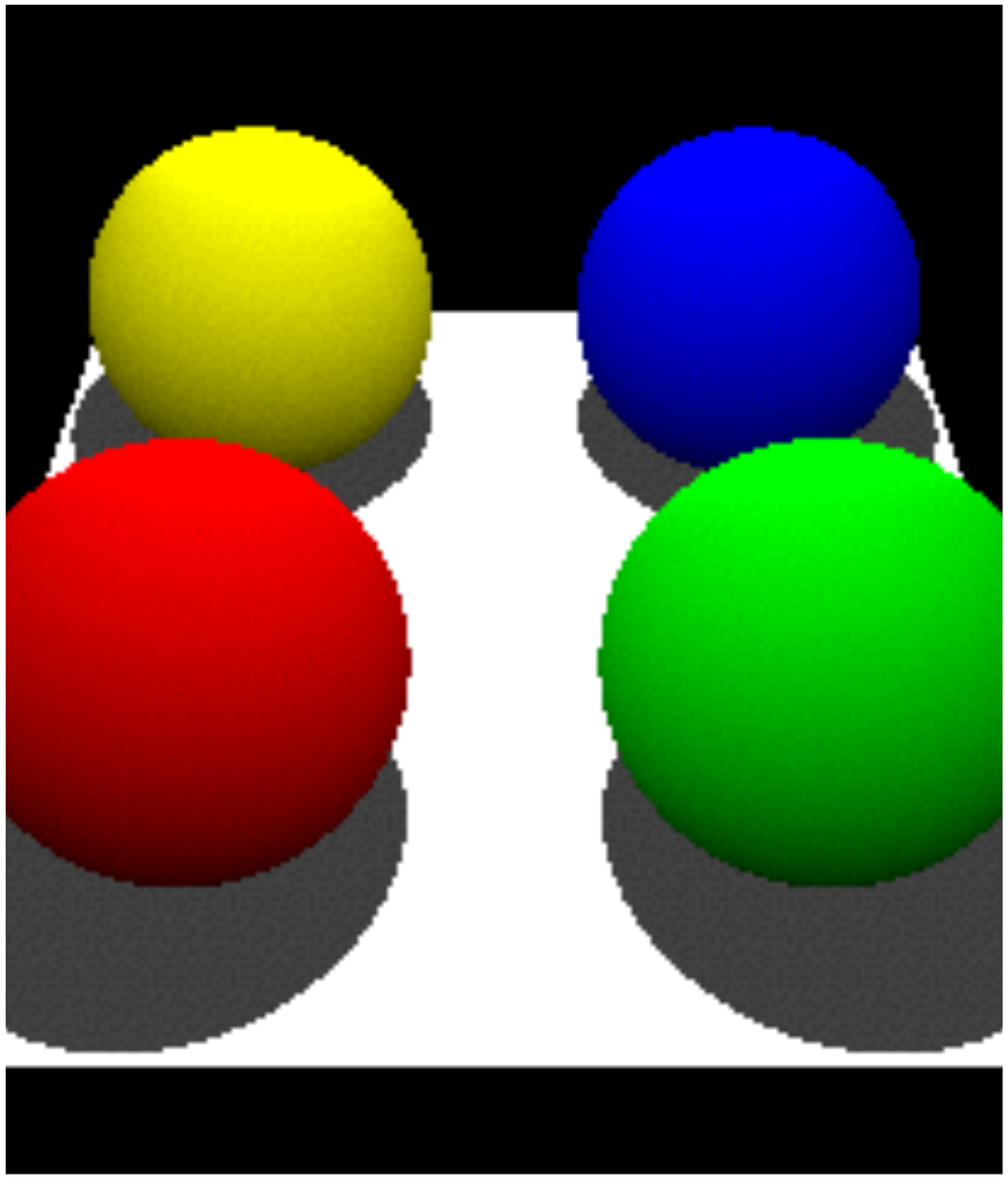
Mise en pratique de la programmation orientée objet

Synthèse d'image par l'algorithme de lancer de rayon

Étudier et enrichir un code source orienté objet.

Ajouter de nouvelles primitives géométriques.

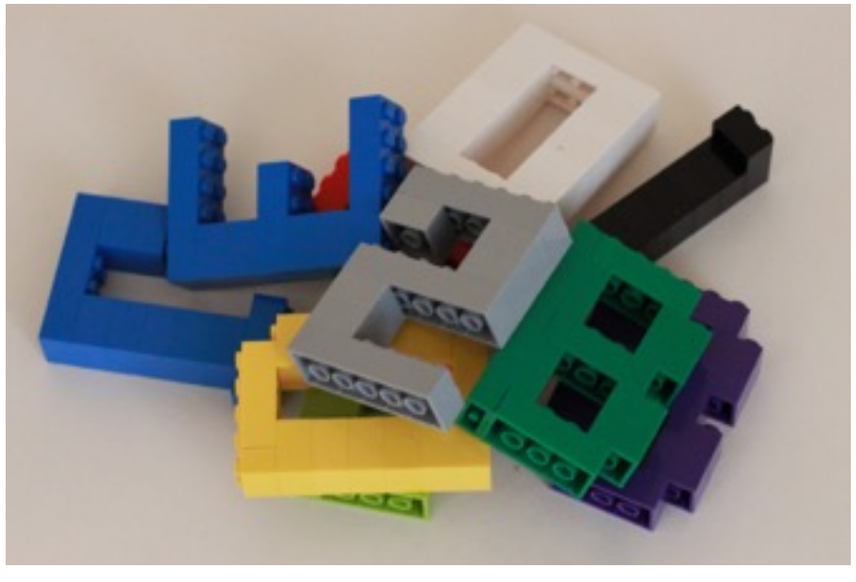
Ajouter la modélisation géométrique des solides (CSG).




JEDI
2018
31

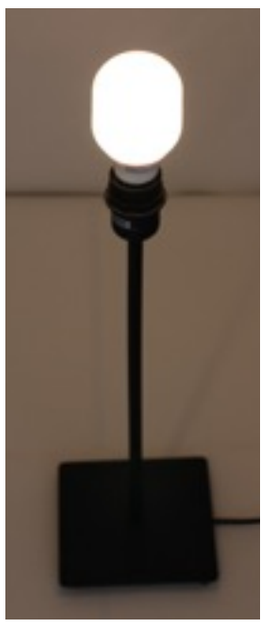
Scène 3D

```
class Scene(object):
    def __init__(self):
        self.camera = None
        self.nodes = []
        self.lights = []

def add(self, object):
    self.nodes.append(object)

def addLight(self, light):
    self.lights.append(light)

def setCamera(self, cam):
    self.camera = cam
```



+

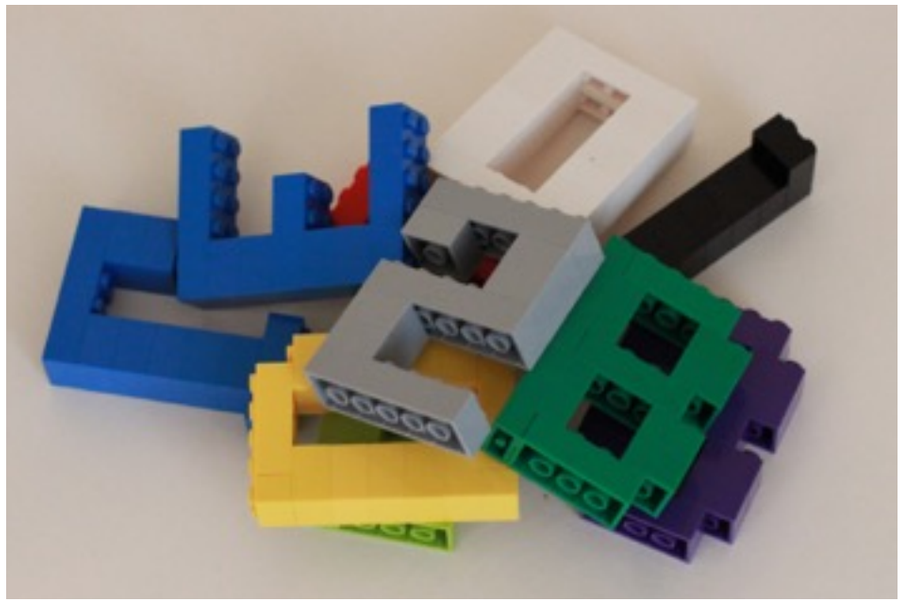
objets



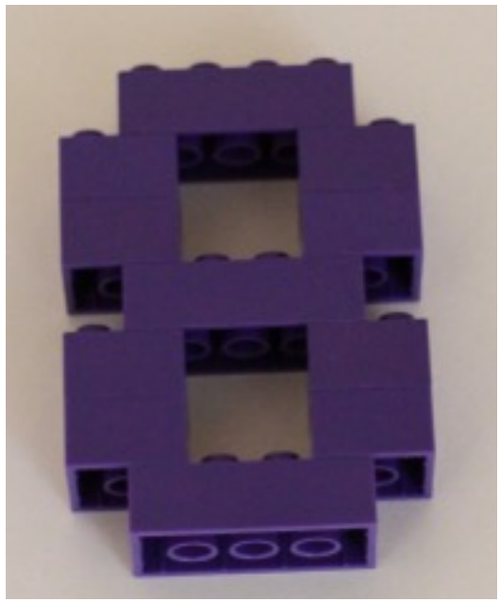
+

lumière observateur

Objets et primitives élémentaires



collection



objet

primitives