Python
Caractéristiques générales

Nicolas Delestre

INSA
ROUEN NORMANDIE

Historique

- Langage créé en 1990 par Guido van Rossum
  - Branche 2.0 à partir de 2000
  - Branche 3.0 à partir de 2008
- S'inspire de plusieurs langages : ABC, Modula, C, Smalltalk, langages de scripts
- Langage open source (GPL), plusieurs implantations (**CPython**, Pypy, Jython, etc.)
- Versions utilisées dans ce cours **3.9**

INSA ROUEN NORMANDIE

Un langage « interprété »

- Deux utilisations possibles de Python :
  1. Depuis un environnement interactif : chaque commande est interprétée avec l'affichage possible d'un résultat
     - python ou python3 : interpréteur de base
     - ipython ou ipython3 : interpréteur étendu (complétion, erreurs plus lisibles, « commandes magiques », etc.)
     - jupyter : interpréteur en ligne (web) très utilisé en science des données
  2. En lançant un script python depuis la ligne de commande
     - Python vérifie l'existence d'un .pyc
     - s'il n'existe ou est plus ancien que le .py, Python compile le code python dans un bytecode
     - Python interprète alors le bytecode

Attention

- Le terme Python désigne quelque fois le langage, quelque fois l'interpréteur

Paradigmes de programmation : introductions ou rappels

Paradigme de programmation

- Un paradigme de programmation est une façon d'aborder informatiquement la résolution d'un problème
- Il propose des concepts et des « outils »
- Un sous paradigme de programmation précise ou ajoute des concepts ou des outils

Paradigme de programmation...

- Imperatif
  - Structuré
- Déclaratif
  - Descriptive
  - Fonctionnel
  - Logique
- Orienté Objet
  - Orienté Classe
  - Prototype
- Événementiel
- Concurrente
- Orienté métaprogrammation
- Orienté pile
- ...

Un langage peut appartenir à plusieurs paradigmes

Paradigmes de programmation orienté objet

Paradigmes de programmation orienté objet

- Le paradigme de la programmation orienté objet repose sur « la définition et l'interaction de brique logiciels appelées objet [qui] représente un concept, une idée ou toute entité du monde physique » (Wikipédia)
- Un objet a un état qui est défini par les valeurs de ces attributs
- Un objet est dit muable si son état peut changer, immuable sinon
- Les méthodes permettent d'interroger l'état d'un objet, de le modifier (lorsque c'est possible) ou d'obtenir, de créer ou de référencer d'autres objets

Paradigmes de programmation orienté classe

- Le paradigme de la programmation orienté classe est un sous paradigme de programmation orienté objet qui indique qu'un objet est créé à partir d'une classe par instanciation
- L'instanciation crée un objet dans un état initial

ROUEN NORMANDIE

Python un langage multi-paradigmes

Avant tout structuré et orienté classe

- Un seul type de sous programme : les fonctions avec passage de paramètre par référence
- Tout est objet : types, fonctions, classes, modules, packages, etc.
- Un objet est l'instance d'une classe qui détermine son type
- Un objet peut être muable ou immuable

Mais aussi du paradigme de la programmation

- Programmation orientée prototype : on peut ajouter, retirer des attributs à un objet
- Fonctionnelle : expression *lambda*, fonction map, filter, etc.
- Concurrente : coroutine
- Métaprogrammation : les classes sont des objets instances de métaclasses

INSA ROUEN NORMANDIE

Affectation et typage

Affectation

- Une variable référence un objet grâce à l'opérateur/instruction d'affectation (=) : la valeur retournée est l'objet référencé
- L'instruction `i = 1` signifie que la variable `i` référence l'objet de type `int` représentant l'entier `1` (et non pas `i` vaut `1`)

Typage

- Typage fort : il n'y a aucun transtypage implicite
- Typage dynamique : le type d'une variable est celui de l'objet dernièrement référencé
- *Duck typing* : le type d'un objet est défini par les messages qu'on peut lui envoyer, par ce qu'il ≪ sait faire ≫, et non pas par le nom de son type
  ≪ *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* ≫ (citation attribuée à James Whitcomb Riley, Wikipédia)
  Le concepteur d'une fonction doit indiquer dans la documentation des paramètres formels leurs types attendus en termes d'opérations/fonctions applicables

Syntaxe 1 / 2

- Fichier obligatoirement en UTF-8 (python 3)
  - on peut utiliser des caractères étendues ≪ accentués, arable, asiatiques, cyrilliques, grecs, etc. ≫ comme identifiant
- L'indentation marque les blocs
- Sensible à la casse
- Commentaires (commençant par un #) et documentations (chaine de caractères, ou *docstring*) sont représentés différemment

INSA ROUEN NORMANDIE

Syntaxe 2 / 2

Bonnes pratiques (`https://www.python.org/dev/peps/pep-0008/`)

- Indentation avec 4 espaces
- Ligne d'au maximum 72 caractères (utilisation de \ pour indiquer la coupure de ligne)
- « Formes » des identifiants varient : en *snake_case* minuscule (variables, fonctions, méthodes, modules, packages), *SNAKE_CASE* en majuscule (constantes) ou en *CamelCase* (classes)
- Une ligne vide entre chaque fonction
- Espaces entre opérateurs (sauf dans les paramètres formels des fonctions) et après les virgules

INSA ROUEN NORMANDIE

Conclusion

Nous avons vu dans ce cours

- Python est un langage interprété
- Python est un langage mutiparadigmes, reposant principalement sur les paradigmes de la programmation structuré et orienté classe
- Tout est objet en Python
- L'affectation en Python référence des objets
- Python est un langage à typage fort, dynamique et qui applique le *duck typing*

INSA ROUEN NORMANDIE

Python
Types de base

Nicolas Delestre

INSA
ROUEN NORMANDIE

Objets, attributs et méthodes

- Tout est objet en Python
- Un objet a un état défini par les ≪ valeurs ≫ de ces attributs
- On accède à un attribut `a` d'un objet `o` à l'aide de la notation pointée `o.a`
- On peut agir (interroger, modifier, obtenir d'autres objets) sur l'objet en envoyant un message à un objet
- On envoie un message (`m`) à un objet à l'aide de la notation pointée avec entre parenthèses (obligatoires) des paramètres effectifs (optionnels) : `o.m([param1, ...])`
- Python cherche alors la méthode (le code python) à invoquer (interpréter)

INSA ROUEN NORMANDIE

Identité vs égalité

- En python une variable référence un objet
- Deux variables `a` et `b` sont considérées comme égales si elles référencent des objets qui sont égaux (même état mais références différentes)
  - `a == b` retourne `True`
  - `a is b` retourne `False`
- Deux variables sont considérées commes identiques si elles référencent le même objet
  - `a == b` retourne `True`
  - `a is b` retourne `True`

INSA ROUEN NORMANDIE

Le type `bool`

- Deux objets immuables à référence unique : `True` et `False`
- Trois opérateurs : `or`, `and` et `not`

Opérateurs de comparaison qui retourne un booléen

`<`, `>`, `<=`, `>=`, `==` (égal), `!=`, `is` (identique) et `is not`

INSA ROUEN NORMANDIE

Nombres 1 / 2

Les types numériques

Types de données immuables

- `int` représentant les entiers signés
  - préfixe des constantes (par défaut représentation décimale) : `0b` ou `0B` pour binaire, `0o` ou `0O` pour octale, `0x` ou `0X` pour notation hexadécimale,
  - `1_000_000`
  - de 0 à 256 références uniques

```
>>> a = 2
>>> b = 1 + 1
>>> a is b
True
```

```
>>> a = 300
>>> b = 300
>>> a is b
False
```

- `float` représentant les flottants signés : `1.`, `1.0`, `1e10`, `1.E-10`
- `complex` représentant les nombres complexes : `2+3j`, `3.j+2`

Opérateurs et fonctions sur les types numériques

- Opérateurs : `+`, `-`, `*`, `/`, `//`, `%`, `**`
- Fonctions : `abs`, `int`, `float`, `complex`, `divmod`, `pow`
- Méthode (uniquement sur les complexes) : `conjugate`
- Opérateurs bits à bits (uniquement sur les `int`) : `~`, `<<`, `>>`

INSA ROUEN NORMANDIE

Sequences 1 / 4

tuple

- Suite immuable d'objets, qui peuvent être muables, de type divers
- Exemples : `1,1.0,"abc",a` ou `(1,1.0,"abc",a)` ou `tuple(s)` avec `s` une séquence

range

- Suite immuable d'entiers ordonnés séparés d'un certains pas (par défaut de 1)
- Syntaxe : `range(fin)`, `range(debut, fin[, pas])`
- L'entier de `fin` n'est pas inclu
- Exemple : `range(10)`

list

- Suite muable d'objets de type divers
- Exemples : `[1,1.0,"abc",a]` ou `list(s)` avec `s` une séquence

str

- Suite immuable de caractères UTF-8
- Constantes peuvent utiliser des simples quotes, doubles quotes ou des triples simples quotes ou triples doubles quotes
- De nombreuses méthodes permettant d'interroger, de découper, de retrouver, de remplacer et de formater une chaîne. À chaque fois elles retournent une valeur (par exemple la nouvelle chaîne calculée) `https://docs.python.org/fr/3/library/stdtypes.html#text-sequence-type-str`
- Mise en forme de chaîne avec l'utilisation de l'opérateur `%` (utilisation d'un tuple si plusieurs valeurs) ou des *f-string* (à partir de la version 3.6)

```
>>> a=1
>>> b="une chaîne"
>>> "a vaut %d et b vaut '%s'" % (a,b)
"a vaut 1 et b vaut 'une chaîne'"
>>> f"a vaut {a} et b vaut {b}"
'a vaut 1 et b vaut une chaine'
```

slice

- Permet de désigner une partie d'une séquence

```
>>> a=tuple(range(0,20,2)) # tuple composé des 10 premiers nombres pairs
>>> a
(0, 2, 4, 6, 8, 10, 12, 14, 16, 18)
```

```
>>> a[slice(None)]
(0, 2, 4, 6, 8, 10, 12, 14, 16, 18)
>>> # d'indice >=0 et <5
>>> a[slice(0,5)]
(0, 2, 4, 6, 8)
>>> a[slice(1,4)]
(2, 4, 6)
>>> a[slice(-2,None)]
(16, 18)
>>> # d'indice >=0 et <5 par pas de 2
>>> a[slice(0,5,2)]
(0, 4, 8)
```

```
>>> a[:]
(0, 2, 4, 6, 8, 10, 12, 14, 16, 18)
>>> # d'indice >=0 et <5
>>> a[0:5]
(0, 2, 4, 6, 8)
>>> a[1:4]
(2, 4, 6)
>>> a[-2:]
(16, 18)
>>> # d'indice >=0 et <5 par pas de 2
>>> a[0:5:2]
(0, 4, 8)
```

Opérations, fonctions, méthodes

opérations `in`, `not in`, `+`, `*`, `[i]`, `[i:j]`, `[i:j:k]`

fonctions `len`, `min`, `max`

méthodes `index(X[, i[, i]])`, `count(x)`

Opérations supplémentaires pour les séquences muables

opérations `s[i] = x`, `s[i:j] = t`, `s[i:j:k] = t`

fonctions `del s[i:j]`, `del s[i:j:k]`

méthodes `append`, `clear`, `copy`, `extend`, `insert`, `pop`, `remove`, `reverse`

INSA ROUEN NORMANDIE

Ensemble

set et frozenset

- Ensemble d'objets *hashables*
- Exemples : `{1,2,3,1}` ou `set(s)` avec `s` une séquence pour obtenir un ensemble muable, ou `frozenset(s)` pour obtenir un ensemble immuable

Opérations, fonctions

fonction	méthode	opérateur
		`in`
		`not in`
`len`		
	`isdisjoint`	
	`issubset`	`<=`
		`<`
	`issuperset`	`>=`
		`>`

fonction	méthode	opérateur
	`union`	`|`
	`intersection`	`&`
	`difference`	`-`
	`symmetric_difference`	`^`

Dictionnaire

dict

- Association de clés et de valeurs
- Les clés doivent être *hashables* (pas possible pour les séquences muables)

Exemples

```
>>> d={1:"a",(1,2):"b"}
>>> d[1]
'a'
>>> d[(1,2)]
'b'
```

Opérations, fonctions, méthodes

opérations `in`, `not in`, `[i]`,

fonction `len`

méthodes `clear`, `copy`, `get`, `items`, `keys`, `pop`, `popitem`, `update`, `values`

Conclusion

Dans ce cours nous avons

- rappelé ce qu'est un objet, un attribut, une méthode et un message
- rapellé la différence entre égalité et identité et les opérateurs associés
- vu que de base Python propose :
  - un type pour représenter les booléens : `bool` (immuable)
  - trois types pour les nombres : `int` (immuable), `float` (immuable), `complex` (immuable)
  - des types pour les séquences : de caractères : `str` (immuable), d'entiers positifs : `range` (immuable), d'objets : `list` (muable), `tuple` (immuable)
  - deux types pour les ensembles mathématiques : `set` (muable), `frozenset` (immuable)
  - un type pour les dictionnaires (ou tableaux associatifs) : `dict` (muable)
- vu que l'on peut désigner une sous partie d'une séquence à l'aide des `slice`
- vu que Python propose du ≪ sucre syntaxique ≫ facilitant la création des listes (`[]`), des tuples (`()`), des ensembles muables (`{}`), des dictionnaires (`{}`) et des slices (`:`)

INSA ROUEN NORMANDIE

Python
Instructions de base

Nicolas Delestre

INSA
ROUEN NORMANDIE

Rappels

Objets, attributs et méthodes

- Tout est objet en Python
- Un objet a un état défini par « les valeurs » de ces attributs
- On accède à un attribut `a` d'un objet à l'aide de la notation pointée (`o.a`)
- On peut agir (interroger, modifier, obtenir d'autres objets) sur l'objet en envoyant un message à un objet
- On envoie un message (`m`) à un objet `o` (ou on invoque, ou appelle, une méthode `m` d'un objet `o`) à l'aide de la notation pointée avec entre parenthèses (obligatoires) des paramètres effectifs (optionnels) : `o.m()`
- Python cherche alors la méthode (le code python) à interpréter

INSA ROUEN NORMANDIE

Affectation

=

- Instruction qui permet de référencer un objet à l'aide d'une variable

- Affectation des références

```
>>> a = 12.5
>>> b = 12.5
>>> c = a
>>> a is b
False
>>> a is c
True
```

- Possibilité de faire plusieurs affectations en une seule fois (utilisation des tuples)

```
>>> a,b = 1,2
>>> a,b = b,a
>>> a
2
>>> a = 1,2
>>> a
(1,2)
```

- L'affectation est aussi une opération, les références retournées sont celles affectées

```
>>> a = b = 1
>>> a
1
>>> b
1
```

Attention

Par abus de langage on dit pour décrire l'instruction `a = 1` que « a vaut 1 » mais on devrait dire que « a référence l'objet `int 1` »

Conditionnelle

if [elif] [else]

- Syntaxe :

```
if condition:
    ...
[elif condition:
    ...
]
[else:
    ...
]
```

```
if anciennete < 6:
    nb_jours = anciennete
elif anciennete < 12:
    nb_jours = 2 * anciennete
else:
    nb_jours = 28
if cadre:
    if age >= 35 and anciennete >= 36:
        nb_jours = nb_jours + 2
    if age >= 45 and anciennete >= 60:
        nb_jours = nb_jours + 4
```

INSA ROUEN NORMANDIE

Itération déterministe 1 / 3

Itérable

- Objet dont on peut parcourir les valeurs
- Les séquences, les ensembles et les dictionnaires sont des itérables (pour les dictionnaires cela permet de parcourir les clés)

`for in [else]`

- Syntaxe :

```
for e in iterable:
    ...
[else:
    ..
]
```

- La partie `else` est exécutée lorsque l'itérable a été parcouru entièrement (pas exécutée si on sort de la boucle à cause d'un `break`)

Itération déterministe 2 / 3

```
>>> for jour in ("lundi", "mardi", "mercredi", "jeudi", \
...              "vendredi", "samedi", "dimanche"):
...     print(jour)
...
lundi
mardi
mercredi
jeudi
vendredi
samedi
dimanche
```

ROUEN NORMANDIE

Itération déterministe 3 / 3

Fonction enumerate

- Syntaxe :

```
for i,e in enumerate(iterable):
    ...
```

```
>>> for numero, jour in enumerate(("lundi", "mardi", "mercredi",\
...                                "jeudi", "vendredi", "samedi",\
...                                "dimanche")):
...     print(f"{numero} {jour}")
...
0 lundi
1 mardi
2 mercredi
3 jeudi
4 vendredi
5 samedi
6 dimanche
```

Itération indéterministe

while [else]

- Syntaxe :

```
while condition:
    ...
[else:
    ...
]
```

- La partie `else` est exécutée lorsque la condition devient fausse (pas exécutée si on sort de la boucle à cause d'un `break`)

INSA ROUEN NORMANDIE

Opérateur de morse

- La version 3.8 de python a introduit l'opérateur de morse qui permet de réaliser des affectations dans les conditions des conditionnelles ou des itérations indéterministes
- Syntaxe : `:=`

Exemple inspiré de `https://www.dad3zero.net/202002/python-walrus-operator/`

```
long = len(ma_sequence)
if long > 10:
    print(f"Vous avez {long} éléments")
```

```
if (long := len(ma_sequence)) > 10:
    print(f"Vous avez {long} éléments")
```

INSA ROUEN NORMANDIE

Autres instructions 1 / 2

Instructions qui peuvent être utiles

- `pass` instruction qui ne fait rien : utile lorsque l'on reporte le développement d'un corps (d'un `if`, `for`, `while` ou une fonction) à plus tard
- `break` instruction qui permet de sortir prématurément d'une itération
- `continue` instruction qui permet de passer à l'itération suivante sans interpréter les instructions qui suivent
- `del` instruction qui permet de supprimer la référence d'une variable

```
>>> a=(1, 2)
>>> del(a)
>>> a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
```

Autres instructions 2 / 2

Instructions qui seront présentées dans de futures cours

- `import` et `from...import` instructions qui permettent d'importer un module, une fonction une classe, etc.
- `try...catch` et `raise` instructions qui permettent de gérer les erreurs (exceptions)
- `return` instruction qui permet de sortir d'une fonction en retournant (optionnellement) la référence vers un objet
- `yield` instruction qui permet de définir des générateurs
- `assert` instruction qui permet d'ajouter une assertion (vérification)
- `global` et `nonlocal` instructions qui permettent de modifier la portée d'une ou de plusieurs variables qui seraient locales sans ces instructions

INSA ROUEN NORMANDIE

Conclusion

Dans ce cours nous avons...

- rappelé que tout est objet en Python et qu'une variable Python référence un objet
- défini le rôle de l'instruction d'affectation
- listé les instructions permettant de coder les schémas conditionnels et itératifs
- listé d'autres instructions dont la plupart seront présentés dans de futurs cours

INSA ROUEN NORMANDIE

Python
Les fonctions

Nicolas Delestre

INSA
ROUEN NORMANDIE

Définition standard d'une fonction 1 / 5

def

- Syntaxe :

```
def nom(param1, param2, ...):
    """ documentation """
    ...
```

- Il n'y a pas de procédure en python, uniquement des fonctions qui retournent `None` lorsque l'instruction `return` n'est pas utilisée ou utilisée avec aucun objet
- Le passage de paramètre est un passage par référence : possibilité de changer l'état de l'objet (si muable) mais il n'est pas possible de changer d'objet (référencé par le paramètre effectif)
- Une variable initialisée dans une fonction est une variable locale (sauf, si grâce à l'instruction `global`, elle est déclarée comme globale)
- Une variable utilisée sans être initialisée est considérée comme globale
- Lors de l'appel d'une fonction, l'association paramètres effectifs $\leftrightarrow$ paramètres formels peut être « positionnel » ou « nommé »

Exemple

```
def est_divisible_par(a, b):
    """ permet de savoir si a est divisible par b"""
    return a % b == 0
```

```
>>> est_divisible_par(39, 3)
True
>>> est_divisible_par(a=39, b=3)
True
>>> est_divisible_par(b=3, a=39)
True
```

```
>>> help(est_divisible_par)


Help on function est_divisible_par in module exemple:

est_divisible_par(a, b)
    permet de savoir si a est divisible par b
```

PEP 257

https://www.python.org/dev/peps/pep-0257/

INSA ROUEN NORMANDIE

- Une fonction peut définir des fonctions locales

Exemple

```
def est_premier(n):
    """permet de savoir si un nombre est premier ou pas"""

    def est_pair(n):
        return est_divisible_par(n, 2)

    if est_pair(n):
        return False
    else:
        for i in range(3, n // 2, 2):
            if est_divisible_par(n, i):
                return False
    return True
```

ROUEN NORMANDIE

- L'instruction `nonlocal` permet, dans une fonction locale, de déclarer qu'une variable est partagée par cette fonction et la fonction « mère »

Exemple

```
def toto():
    def tata():
        nonlocal x
        x = 1
    x = 0
    tata()
    return x
```

INSA ROUEN NORMANDIE

Annotation 1 / 2

def

- Syntaxe :

```
def nom(param1: annot1, param2: annot2, ...) -> annotRetour:
    """ documentation """
    ...
```

- Les annotations (à partir de python 3.4) sont optionnelles et doivent être des expressions python (souvent les types attendus, mais cela peut poser problème pour les classes en cours de définition)
- Les informations fournies par les annotations peuvent servir :
  - aux développeurs utilisateurs de la fonction (complète la documentation)
  - aux environnements de développement (où IDE)
  - à des vérificateurs statiques de type, tel que *mypy* (cf. `http://mypy-lang.org/`)

INSA ROUEN NORMANDIE

Exemple

```
def est_premier(n: int) -> bool:
    """permet de savoir si un nombre est premier ou pas"""
    if n % 2 == 0:
        return False
    else :
        for i in range(3, n // 2, 2):
            if n % i == 0:
                return False
    return True
```

En utilisant ≪ des valeurs ≫ par défaut

- Syntaxe :

```
def nom(param1, ..., paramOpt1 = val1, paramOpt2 = val2...):
    """ documentation """
    ...
```

- Les paramètres formels ayant une valeur par défaut deviennent optionnels
- Dès qu'un paramètre formel à une valeur par défaut, les suivants doivent aussi en avoir

Attention

Il est conseillé d'utiliser des objets immuable pour éviter tout problème

INSA ROUEN NORMANDIE

Définition de fonction d'arité variable 2 / 6

Exemple

```
def somme_naturels(fin, debut=1, pas=1):
    res = 0
    for i in range(debut,fin,pas):
        res = res + i
    return res
```

```
def somme_naturels(fin: int, debut: int=1, pas: int=1) -> int:
    res = 0
    for i in range(debut,fin,pas):
        res = res + i
    return res
```

```
>>> somme_naturels(10)
45
>>> somme_naturels(10,5)
35
>>> somme_naturels(10,pas=2)
25
>>> somme_naturels(debut=1,fin=10,pas=2)
```

En utilisant *args

- Syntaxe :

```
def nom(param1, ..., *args):
    """ documentation """
    ...
```

- Cela signifie que le nombre de paramètres effectifs non nommés peut être en nombre variable
- `*args` doit être déclaré après les paramètres formels nommés
- Lors de l'appel, il doit y avoir une valeur pour tous les paramètres nommés
- `args` est un tuple
- On peut transformer une séquence en une suite variable de paramètres effectifs grâce à l'opérateur `*`, par exemple avec `l = (1, 2, 3)`

  `foo(*l)` est équivalent à `foo(1, 2, 3)`

Définition de fonction d'arité variable 4 / 6

Exemple

```
def somme_nombres(debut, fin, *args):
    res = 0
    for i in args[debut:fin]:
        res = res + i
    return res
```

```
>>> somme_nombres(0,3,10,20,30,40,50)
60
>>> somme_nombres(0,3,10,20,30,40,50,60,70)
60
>>> somme_nombres(0,7,10,20,30,40,50,60,70)
280
>>> somme_nombres(0,3,*[10,20,30,40,50,60,70])
60
```

En utilisant `**kwargs`

- Syntaxe :

```
def nom(param1, ..., **kwargs):
    """ documentation """
    ...
```

- Cela signifie que le nombre de paramètres effectifs nommés peut être en nombre variable (différents des paramètres formels)
- `kwargs` est un dictionnaire
- `**kwargs` doit être déclaré en dernier, après `*args`
- On peut transformer un dictionnaire ayant des clés de type `str` en une suite de paramètres effectifs nommés grâce à l'opérateur `**`

ROUEN NORMANDIE

Définition de fonction d'arité variable 6 / 6

Exemple

```
1  def valeur_d_une_cle(dic, cle, valeur_si_non_present):
2      if dic:
3          if cle in dic:
4              return dic[cle]
5          else:
6              return valeur_si_non_present
7      else:
8          return valeur_si_non_present
9
10 def somme_nombres_v2(*args, **kwargs):
11     debut = valeur_d_une_cle(kwargs,'debut',0)
12     fin = valeur_d_une_cle(kwargs,'fin',len(args))
13     res = 0
14     for i in args[debut:fin]:
15         res = res + i
16     return res
```

```
>>> somme_nombres_v2(10,20,30,40)
100
>>> somme_nombres_v2(10,20,30,40,debut=1)
90
>>> somme_nombres_v2(10,20,30,40,debut=1,fin=3)
50
>>> somme_nombres_v2(10,20,30,40,fin=3)
60
```

Instruction `pass`

- Lors de la définition d'une fonction, la signature doit **obligatoirement** être suivie de la documentation et/ou du code
- L'instruction `pass` est une instruction qui ne fait rien
- Elle est utilisée lorsque l'on veut définir une fonction sans documentation qui ne fait rien (on reporte à plus tard son développement)

Exemple

```
def fonction_qui_ne_fait_rien():
    pass
```

INSA ROUEN NORMANDIE

Conclusion

Dans ce cours nous avons vu

- comment développer une fonction :
  - avec ou sans annotation,
  - en utilisant des variables locales ou globales, voire des variables non locales pour les fonctions locales
- comment appeler une fonction avec des liaisons paramètres effectifs $\leftrightarrow$ paramètres formels, positionnelles ou nommées
- comment développer des fonctions d'arité variable
- les opérateurs
  - `*` qui permet de transformer toute séquence en paramètres effectifs positionnels
  - `**` qui permet de transformer tout dictionnaire en paramètres effectifs nommés
- l'instruction `pass` qui permet de créer des fonctions « vides »

ROUEN NORMANDIE

Python
Modules, scripts, paquets

Nicolas Delestre

INSA
ROUEN NORMANDIE

Organisation générale d'un fichier .py

- L'encodage du fichier est obligatoirement utf-8
- Le fichier est composé de plusieurs parties
  1. Le rôle du fichier sous forme de *docstring* qui sera utilisé par help()
  2. Des métadonnées qui seront utilisées par `help()` : `__author__` (str), `__contact__` (sequence de `str`), `__credits__` (sequence de `str`), `__maintainer__` (`str`) `__email__` (`str`) `__version__` (`str`), `__copyright__` (`str`), `__date__` (`str` au format ISO 8601)
  3. Les déclarations/définitions :
     - Variables globales (en *SNAKE_CASE* majuscule)
     - Classes
     - Fonctions
     - Le code à exécuter (équivalent du *main* dans d'autres langages)

INSA ROUEN NORMANDIE

Script

- Suite de définitions et d'instructions d'un fichier `.py` qui représente un programme

```
$ python3 nom_du_script.py
```

- Grâce au *shebang* (`#!`) on peut indiquer au système d'exploitation UNIX quel interpréteur exécuté lorsque le script est utilisé avec les droits en exécution. Le première ligne du script doit alors être :

  #!/usr/bin/env python3

```
$ ./nom_du_script.py
```

- Suite de définitions et d'instructions d'un fichier `.py` qui peuvent être importées (dans un autre module ou dans un script)
- Les définitions sont propres au module
- Les instructions en dehors de toute fonction ou classe sont interprétées au premier `import` (elles servent à initialiser certaines variables globales au module)
- Un module peut être utilisé en tant que script.
  `if __name__== "__main__":` permet de conditionner l'interprétation d'instructions dans ce cas

INSA ROUEN NORMANDIE

fibo.py

```
#!/usr/bin/env python3
"""
Exemple issu du tutoriel de python: https://docs.python.org/
"""

__author__ = "Nicolas Delestre"
__contact__ = "Nicolas Delestre"
__copyright__ = "Copyleft"
__credits__ = ["Nicolas Delestre"]
__license__ = "GPL"
__version__ = "1.0.1"
__maintainer__ = "Nicolas Delestre"
__email__ = "nicolas.delestre@insa-rouen.fr"
__status__ = "Production"

def fib(n):
    """ retourne les nombres de fibonacci jusqu'au rang n """
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result

if __name__ == "__main__":
    import sys
    print(fib(int(sys.argv[1])))
```

Comment importer un module ?

- `import nom_du_module` : l'utilisation d'un élément du module nécessite de préfixer cet élément par le nom du module (notation pointée)
- `from nom_du_module import element` : l'utilisation de l'élément est alors direct (sans faire référence au module)
  - l'utilisation de `*` en lieu et place de l'élément permet d'importer toutes des définitions d'un module (à éviter)
- le suffixe `as` permet de renommer localement un module importé ou l'élément du module importé

INSA ROUEN NORMANDIE

Attention à la casse des caractères dans le nommage des modules !

- La PEP8 (`https://www.python.org/dev/peps/pep-0008/`) préconise d'utiliser le *snake_case* pour nommer les modules

Comment l'interpréteur Python trouve-t-il le fichier `.py` nécessaire ?

- Lorsqu'un module est importé (par exemple `import un_module`), l'interpréteur Python recherche le fichier correspondant (pour l'exemple `un_module.py`) dans la liste des répertoires référencés par la variable `sys.path`
- `sys.path` contient
  1. le répertoire courant
  2. les répertoires présents dans la variable d'environnement `PYTHONPATH`
  3. les répertoires systèmes Python

ROUEN NORMANDIE

```
$ python3 fibo.py 10
[1, 1, 2, 3, 5, 8]
$ chmod +x fibo.py
$ ./fibo.py 10
[1, 1, 2, 3, 5, 8]
$ python3
Python 3.5.2 (default, Nov 17 2016, 17:05:23)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import fibo
>>> fibo.fib(10)
[1, 1, 2, 3, 5, 8]
>>> import fibo as fb
>>> fb.fib(10)
[1, 1, 2, 3, 5, 8]
>>> from fibo import fib as fibonacci
>>> fibonacci(10)
[1, 1, 2, 3, 5, 8]
```

Paquet (ou *Package*) 1 / 4

- Organisation hiérarchique de paquets et de modules
- `import` permet de désigner un module en donnant le chemin des paquets (notation pointée)
- `from..import` permet d'importer
  - un module : entre le `from` et l'`import` il n'y a qu'un chemin de paquets
  - un élément d'un module : entre le `from` et l'`import` il y a un chemin de paquets se terminant par le module
- Les paquets sont représentés par :
  - un répertoire, le contenu du paquet (paquet ou module) est dans le répertoire
  - un fichier __init__.py (obligatoire) dans le répertoire, qui contient une suite d'instructions qui permet d'initialiser le paquet (exécutée lors du premier import).
    C'est ici qu'est contrôlé le `from..import *` pour un paquet (initialisation de la variable `__all__` du paquet)

Exemple issu du tutoriel Python

```
sound/                          Top-level package
      __init__.py               Initialize the sound package
      formats/                  Subpackage for file format conversions
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
      effects/                  Subpackage for sound effects
              __init__.py
              echo.py
              surround.py
              reverse.py
      filters/                  Subpackage for filters
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
```

```
$ cat sound/__init__.py
$ cat sound/formats/__init__.py
$ cat sound/effects/__init__.py
#!/usr/bin/env python3
__all__ = ["echo","surround","reverse"]
$ cat sound/filters/__init__.py
#!/usr/bin/env python3
__all__ = ["equalizer"]
$ cat sound/formats/wavread.py
#!/usr/bin/env python3
def wavread():
    print("wavread")
```

INSA ROUEN NORMANDIE

Exemple issu du tutoriel Python

1

```
>>> import sound.effects.echo
>>> sound.effects.echo.echo()
echo
>>> from sound.effects import echo
>>> echo.echo()
echo
>>> from sound.effects.echo import echo
>>> echo()
echo
```

2

```
>>> from sound.formats import *
>>> wavread.wavread()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'wavread' is not defined
```

3

```
>>> from sound.effects import *
>>> echo.echo()
echo
>>> surround.surround()
surround
>>> reverse.reverse()
reverse
```

4

```
>>> from sound.filters import *
>>> equalizer.equalizer()
equalizer
>>> vocoder.vocoder()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'vocoder' is not defined
```

Import relatif

- Lorsque le module d'un paquet nécessite l'import d'un paquet ou d'un module d'une autre branche du même procjet, on peut utiliser des imports relatifs (`.` pour le pauqte courant, `..` pour le paquet parent)

Exemple issu du tutoriel Python

```
sound/
      __init__.py
      formats/
              __init__.py
              ...
      effects/
              __init__.py
              echo.py
              surround.py
              ...
      filters/
              __init__.py
              equalizer.py
              ...
```

Depuis le module `surround.py`

```
from . import echo
from .. import formats
from ..filters import equalizer
```

Conclusion

Nous avons vu dans ce cours

- qu'un fichier .py se nomme un module
- qu'un module peut être un script
- que l'on peut organiser hiérarchiquement le code en paquet, qui contiennent des paquets ou des modules
- que l'instruction `import` permet de charge un module
- que l'instruction `from .. import` permet de charger un module d'un packahe ou l'élément d'un module
- que le suffixe `as` permet de renommer localelemnt un module ou l'élément d'un module

INSA ROUEN NORMANDIE

Python
Expressions Lambda

Nicolas Delestre

INSA
ROUEN NORMANDIE

Définition

- Une expression lambda est une fonction anonyme avec une seule instruction qui est une unique expression
  - Non utilisation du mot clé `return`, la valeur de l'expression est la valeur retournée
- Elle est utilisée comme :
  - valeur par défaut d'un paramètre formel
  - paramètre effectif
  - exceptionnellement comme valeur d'une affectation
- Elle évite de créer des fonctions à usage unique

INSA ROUEN NORMANDIE

Syntaxe

```
lambda [param1, param2, ...] : expression
```

Équivalence

```
foo = lambda parametres : expression
```

est équivalent à

```
def foo(parametres):
    return expression
```

INSA ROUEN NORMANDIE

Exemple 1 : ordre de tri

```
def trier(l, doivent_etre_echanges=lambda x, y: x > y):
    est_trie = False
    while not est_trie:
        est_trie = True
        for i in range(len(l)-1):
            if doivent_etre_echanges(l[i], l[i+1]):
                l[i], l[i+1] = l[i+1], l[i]
                est_trie = False
```

```
>>> l=[3,2,8,1,5,98,3]
>>> trier(l)
>>> l
>>> [1, 2, 3, 3, 5, 8, 98]
>>> trier(l,lambda x,y : x < y)
>>> l
>>> [98, 8, 5, 3, 3, 2, 1]
```

Exemple 2 : généralisation du tri 1 / 2

```
def trier(l,
          doivent_etre_echanges=lambda x, y: x > y,
          valeur_a_comparer=lambda x: x):
    est_trie = False
    while not est_trie:
        est_trie = True
        for i in range(len(l)-1):
            if doivent_etre_echanges(valeur_a_comparer(l[i]),
                                     valeur_a_comparer(l[i+1])):
                l[i], l[i+1] = l[i+1], l[i]
                est_trie = False
```

INSA ROUEN NORMANDIE

Exemple 2 : généralisation du tri 2 / 2

```
>>> l=[complex(3,1), complex(1,3), complex(2,2)]
>>> trier(l)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 8, in trier
  File "<stdin>", line 2, in <lambda>
TypeError: '>' not supported between instances of 'complex' and 'complex'
>>> trier(l, valeur_a_comparer=lambda z:z.real)
>>> l
[(1+3j), (2+2j), (3+1j)]
>>> trier(l, valeur_a_comparer=lambda z:z.imag)
>>> l
[(3+1j), (2+2j), (1+3j)]
>>> from math import sqrt
>>> trier(l, valeur_a_comparer=lambda z:sqrt(z.real**2+z.imag**2))
>>> l
[(2+2j), (3+1j), (1+3j)]
```

INSA ROUEN NORMANDIE

Conclusion

Les expressions lambda :

- sont des fonctions anonymes très simples
- sont issues de la programmation fonctionnelle
- permettent de généraliser des algorithmes
- permettent d'éviter de créer des fonctions peu utiles

INSA ROUEN NORMANDIE

Python
Map, filter et *comprehension*

Nicolas Delestre

INSA
ROUEN NORMANDIE

Contexte

Créer une nouvelle liste à partir d'une séquence (*source*)

- assez courant
- algorithme fondé sur l'initialisation d'une liste résultat (à vide) et sur le parcours de la séquence donnée avec le complètement de la liste résultat en appliquant une formule ou en sélectionnant des éléments

Pour appliquer une formule

```
def mettre_au_carre_v0(l):
    res = []
    for n in l:
        res.append(n ** 2)
    return res
```

Pour filtrer des éléments

```
def selectionner_pairs_v0(l):
    res = []
    for n in l:
        if n % 2 == 0:
            res.append(n)
    return res
```

Généralisation

Pour appliquer une formule

```
def appliquer(l,
              calcul=lambda x: x):
    res = []
    for n in l:
        res.append(calcul(n))
    return res


def mettre_au_carre_v1(l):
    return appliquer(l,
                     lambda n: n**2)
```

Pour filtrer des éléments

```
def selectionner(l,
                 a_garder=lambda x: True):
    res = []
    for n in l:
        if a_garder(n):
            res.append(n)
    return res


def selectionner_pairs_v1(l):
    return selectionner(l,
                        lambda n: n % 2 == 0)
```

INSA ROUEN NORMANDIE

map et filter 1 / 2

map

- permet d'appliquer une fonction (souvent une expression lambda) à chaque élément d'une séquence
- syntaxe : `map(fonction, séquence)`
- on obtient un objet itérable tels que les éléments sont calculés à la demande (*yielding iterator*)

```
def mettre_au_carre_v2(l):
    return list(map(lambda n: n**2, l))
```

INSA ROUEN NORMANDIE

map et filter 2 / 2

filter

- permet de sélectionner les éléments d'une séquence , ceux pour lesquels une condition est vrai
- syntaxe : `filter(fonction, liste)`
- comme `map`, `filter` retourne un objet, un *yielding iterator*

```
def selectionner_pairs_v2(l):
    return list(filter(lambda n: n % 2, l))
```

INSA ROUEN NORMANDIE

≪ Compréhension ≫ de séquence

- syntaxe introduite avec python 2 :
  - `[foo(e) for e in sequence]`
  - `(foo(e) for e in sequence)`
  - `{foo(e) for e in sequence}`
  - `{e:foo(e) for e in sequence}`
- équivalente (en terme de fonctionnalité) aux fonctions `map` et `filter`
- plus *pythonic*

```
def mettre_au_carre_v3(l):
    return [n**2 for n in l]

def selectionner_pairs_v3(l):
    return [n for n in l if n % 2 == 0]
```

Performances 1 / 2

```
def benchmark_fonctions_un_parametre_naturel(nom_module, nom_fonctions, taille=10000000):
    for nom_fonction in nom_fonctions:
        code = f"from {nom_module} import {nom_fonction} as m; l=range({taille}); list(m(l))"
        print(f"{nom_fonction}, {timeit.timeit(code, number=1)}")

def benchmark_mettre_au_carre(taille=10000000):
    benchmark_fonctions_un_parametre_naturel("mapFilterComprehension", ["mettre_au_carre_v%d" % i for
        i in range(4)], taille)

def benchmark_selectionner_pairs(taille=10000000):
    benchmark_fonctions_un_parametre_naturel("mapFilterComprehension", ["selectionner_pairs_v%d" % i
        for i in range(4)], taille)

if __name__ == "__main__":
    benchmark_mettre_au_carre()
    benchmark_selectionner_pairs()
```

ROUEN NORMANDIE

Performances 2 / 2

```
$ python mapFilterComprehension.py
mettre_au_carre_v0, 3.116361833992414
mettre_au_carre_v1, 3.7307979429897387
mettre_au_carre_v2, 3.257178820000263
mettre_au_carre_v3, 2.8759335440117866
selectionner_pairs_v0, 0.714096306997817
selectionner_pairs_v1, 1.249474090989679
selectionner_pairs_v2, 0.8969143869762775
selectionner_pairs_v3, 0.5892258970125113
```

INSA ROUEN NORMANDIE

Conclusion

- Quatre façons d'appliquer un traitement ou de sélectionner des éléments d'une séquence :
  - algorithmes classiques
  - algorithmes classiques généralisés : `appliquer`, `selectionner`
  - fonctions de base de python : `map` et `filter`
  - compréhension
- Favoriser les compréhensions car :
  - plus lisibles, plus *pythonic*
  - elles peuvent produire différents types de séquences
  - plus rapides

INSA ROUEN NORMANDIE

Python
`yield`

Nicolas Delestre

INSA
ROUEN NORMANDIE

Itération sans séquence

Concept

- Python permet grâce à l'instruction `yield` de créer des fonctions dont leurs résultats puissent être utilisés comme des séquences sans qu'ils ne le soient
- Cela permet :
  - de ne calculer que les éléments réellement utilisés (gain en temps et en espace mémoire)
  - de pouvoir créer des séquences « infinies »

Instruction `yield`

- équivalent d'un `return` qui retourne une valeur
- au prochain élément demandé, la fonction reprend où elle s'était arrêtée, jusqu'au prochain `yield` ou à la fin de la fonction
- dans 99% du temps le `yield` est au sein d'une itération (finie ou infinie)
- la fonction retourne en fait un « générateur »

Exemple : une fonction qui retourne les nombres premiers 1 / 3

est_premier

```
def est_premier(n):
    if n % 2 == 0:
        return False
    else:
        borne = int(numpy.sqrt(n)) + 1
        for i in range(3, borne, 2):
            if n % i == 0:
                return False
        return True
```

nombres_premiers

```
1  def nombres_premiers(borne_max = None):
2      yield 2
3      i = 3
4      while not borne_max or i <= borne_max:
5          if est_premier(i):
6              yield i
7          i = i + 2
```

Exemple : une fonction qui retourne les nombres premiers 2 / 3

Affichage des nombres premiers inférieurs à 30

```
>>> for i in nombres_premiers(30):
    print(i)
   ...:
2
3
5
7
11
13
17
19
23
29
>>>
```

INSA ROUEN NORMANDIE

Exemple : une fonction qui retourne les nombres premiers 3 / 3

Calcul du premier nombre premier superieur à *n*

```
def premier_nombre_premier_superieur_a(n):
    for i in nombres_premiers():
        if i >= n:
            return i
```

```
>>> premier_nombre_premier_superieur_a(101548)
101561
>>>
```

INSA ROUEN NORMANDIE

Conclusion

- L'instruction `yield` permet de créer des itérateurs qui calculent les valeurs « à la demande »
- Python s'inspire du langage CLU (1977)
- On retrouve cette instruction dans quelques autres langages tels que Javascript, Ruby ou C#

INSA ROUEN NORMANDIE

Python
Les classes

Nicolas Delestre

INSA
ROUEN NORMANDIE

Classe

Déclaration

- Syntaxe :

```
class NomClasse[(SuperClasse,SuperClasse2,...)]:
""" Documentation """
    def __init__(self[, param1, param2, ...]):
        """ Documentation """
        self.attr1 = ...
        self.attr2 = ...

    def m1(self[, param1, param2, ...]):
        ...
```

- On définit une classe dans un module (un module peut contenir plusieurs classes). On importe une classe, comme on importe une fonction
- L'identifiant d'une classe est en *CamelCase*
- La classe au sommet de l'héritage est la classe `object` qui n'est pas précisée si c'est la superclasse de la classe à définir
- Il n'y a pas de classe abstraite

Attributs, méthodes

- C'est le faite d'affecter une valeur à un attribut (affectation), que l'attribut est ajouté à l'instance
- La bonne pratique veut que l'*initialiseur*[a] (`__init__`) initialise tous les attributs d'instance
- Il n'y a pas de mot clé pour la visibilité des méthodes ou des attributs, mais si l'identifiant commence par un :
  - `_` alors il doit être considéré comme privé (il n'apparaît pas dans la documentation, `help`)
  - `__` alors il est privé et non accessible (sans faire d'introspection)
- Comme les paramètres formels ne sont pas typés, il n'y a pas de surcharge (polymorphisme)

a. très souvent dénommé, à tord, *constructeur*. Le véritable constructeur est la méthode `__new__` que l'on ne redéfinit que si on fait de la méta-programmation

Classe

Classe Point2D (sans la documentation) du module point.py

```
 5 class Point2D:
 6     def __init__(self, x: float, y: float, identifiant: str=None):
 7         self._x = x
 8         self._y = y
 9         self._id = identifiant
10
11     def get_x(self) -> float:
12         return self._x
13
14     def set_x(self, x: float):
15         self._x = x
16
17     def get_y(self) -> float:
18         return self._y
19
20     def set_y(self, y: float):
21         self._y = y
22
23     def get_id(self) -> str:
24         return self._id
```

Objet

Création d'un objet

- On crée un objet en utilisant l'identifiant de la classe suivi, entre parenthèses, des paramètres effectifs à donner à l'initialiseur (pour les paramètres formels qui suivent le `self`)

```
>>> from point import Point2D
>>> pt1 = Point2D(1,2)
```

Invocation d'une méthode

- On invoque habituellement une méthode `m` sur un objet `o` en utilisant la notation pointée classique : `o.m(...)`
- On peut aussi invoquer une méthode comme on appelle une fonction, en préfixant l'identifiant de la méthode par l'identifiant de la classe et en mettant en premier paramètre effectif l'objet

```
>>> pt1.get_x()
1
>>> Point2D.get_x(pt1)
1
```

Héritage

Principe

- Python accepte l'héritage multiple
- On peut redéfinir le comportement des méthodes (polymorphisme par héritage) : on accède à la méthode de la super classe en l'invoquant à l'aide de la deuxième notation
- La bonne pratique veut que l'on désigne la super classe à l'aide de la fonction `super`
  Cf. https://stackoverflow.com/questions/42413670/whats-the-difference-between-super-and-parent-class-name

Classe `Point3D` (sans la documentation) du module `point.py`

```
26 class Point3D(Point2D):
27     def __init__(self, x: float, y: float, z: float, identifiant: str=None):
28         super().__init__(x, y, identifiant)
29         self._z = z
30 
31     def get_z(self) -> float:
32         return self._z
33 
34     def set_z(self, z: float):
35         self._z = z
```

Liaison dynamique de méthode

Principe

- Python réalise la liaison dynamique de méthode : la méthode sélectionnée suite à l'invocation d'une méthode est toujours celle qui est la plus proche (au sens de l'héritage) de la classe de l'objet

```
3  class A:
4      def m1(self):
5          print("m1 de A")
6          self.m2()
7
8      def m2(self):
9          print("m2 de A")
10
11 class B(A):
12     def m2(self):
13         print("m2 de B")
```

```
>>> b=B()
>>> b.m1()
m1 de A
m2 de B
>>> a=A()
>>> a.m1()
m1 de A
m2 de A
```

Algorithme de sélection de méthode dans le cas d'un héritage multiple

- L'algorithme utilise un tri topologique tel que les classes sont étudiées dans l'ordre de déclaration de l'héritage multiple
- L'attribut `__mro__` (*method resolution order*) permet de connaître la liste issue de ce tri

```
class A1:
    def m(self):
        print("m de A1")


class A2:
    def m(self):
        print("m de A2")

class B(A1, A2):
    pass
```

```
>>> B().m()
m de A1
```

```
class A1:
    def m(self):
        print("m de A1")


class A2:
    def m(self):
        print("m de A2")

class B(A2, A1):
    pass
```

```
>>> B().m()
m de A2
```

```
class A:
    def m(self):
        print("m de A")

class B1(A):
    pass

class B2(A):
    def m(self):
        print("m de B2")

class C(B1, B2):
    pass
```

```
>>> C().m()
m de B2
```

```
>>> C.__mro__
(<class 'heritage_multiple_3.C'>, <class 'heritage_multiple_3.B1'>,
<class 'heritage_multiple_3.B2'>, <class 'heritage_multiple_3.A'>, <class 'object'>)
```

Retour sur `__init__`

- Le comportement de la méthode `__init__` est classique

```
class A:
    def __init__(self, a):
        self._a = a

class B(A):
    pass

class C(A):
    def __init__(self, c):
        self._c = c

class D(A):
    def __init__(self, a, d):
        super().__init__(a)
        self._d = d
```

```
>>> a=A(1)
>>> a._a
1
>>> b=B(1)
>>> b._a
1
>>> c=C(1)
>>> c._a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'C' object has no attribute '_a'
>>> d=D(1,2)
>>> d._a
1
>>> d._d
2
```

Conclusion

- Comment on définit une classe
- Python réalise de la liaison dynamique de méthode
- Python accepte l'héritage multiple avec une recherche des attributs et méthodes à l'aide d'un tri topologique
- L'initialiseur `__init__` est une méthode comme les autres

INSA ROUEN NORMANDIE

Python
Les propriétés

Nicolas Delestre

INSA
ROUEN NORMANDIE

Contexte

Getter, *setter*, pas très *pythonic*

- Les *setter* et *getter* permettent l'encapsulation :
  - séparer la façon dont est conçue la classe de la façon dont sont utilisés les objets (pouvoir modifier la conception tout en ayant la même façon de les utiliser)
  - vérifier que l'utilisation est bonne (et levée des *exceptions* si besoin)
- Mais pas très *pythonic* (faire confiance à l'utilisateur)

Les propriétés (*properties*)

- faire croire à l'utilisateur qu'il accède directement aux attributs
- être *pythonic* tout en préservant l'encapsulation

```
>>> pt = Point2D(1,2)
>>> pt.get_x()
1
>>> pt.set_x(3)
```

$\Rightarrow$

```
>>> pt = Point2D(1,2)
>>> pt.x
1
>>> pt.x = 3
```

INSA ROUEN NORMANDIE

Comment déclarer une propriété ?

Première syntaxe

```
attr = property(getter[, setter[, deleter,[ doc]]])
```

Mise à jour de la classe Point2D du module point.py

```
5 class Point2D:
6     def __init__(self, x, y, identifiant=None):
7         self._x = x
8         self._y = y
9         self._id = identifiant
10
11    def _get_x(self):
12        return self._x
13
14    def _set_x(self, x):
15        self._x = x
16
17    x = property(_get_x, _set_x, doc="l'abscisse du point 2D")
```

Comment déclarer une propriété ?

Seconde syntaxe (utilisation des décorateurs)

```
@property
def attr(self):
    ...
@attr.setter
def attr(self, valeur):
    ...
@attr.deleter
def attr(self):
    ...
```

Attention

Ce n'est pas de la surcharge syntaxique (cf. cours sur les décorateurs)

Mise à jour de la classe `Point2D` du module `point.py` (suite)

```
19    @property
20    def y(self):
21        return self._y
22
23    @y.setter
24    def y(self, y):
25        self._y = y
```

```
28    @property
29    def id(self):
30        return self._id
```

Conclusion

Nous avons dans ce cours

- L'objectif des propriétés en python ?
- Les deux syntaxes de déclaration des propriétés

Principe général de python

Le concepteur/développeur d'une API écrit plus de code pour simplifier la vie de l'utilisateur

INSA ROUEN NORMANDIE

Python
Méthodes spéciales

Nicolas Delestre

INSA
ROUEN NORMANDIE

Les méthodes spéciales

- Ensemble de méthodes systèmes à définir ou redéfinir pour :
  - **définir le comportement de l'interpéteur au regard des objets (par exemple comment les représenter textuellement)**
  - **de pouvoir utiliser des opérateurs ou fonctions standards avec les objets**
  - définir le comportement des objets dans certaines situations (comme par exemple dans les dictionnaires)
- Ces méthodes ont comme identifiant `__XX__`
  - `__init__` est la méthode spéciale exécutée pour initialiser une nouvelle instance

INSA ROUEN NORMANDIE

Les méthodes `__repr__` et `__str__`

La méthode `__repr__`

- Appelée par la fonction `repr`
- C'est la méthode qui est exécutée lorsque l'interpréteur affiche l'objet
- Représentation textuelle formelle (« informatique ») d'un objet (pourrait être utilisée pour créer un nouvel objet)

Sans redéfinition de `__repr__`

```
>>> class A:
    ....pass
>>> a=A()
>>> a
<__main__.A object at 0x7f13bbd4d9d0>
```

Classe `Point2D` du module `point.py`

```
32    def __repr__(self):
33        if self.id:
34            return f"Point2D({self.x}, {self.y}, '{self.id}')"
35        return f"Point2D({self.x}, {self.y})"
```

```
>>> pt=Point2D(1,2,'A')
>>> pt
Point2D(1, 2, 'A')
```

```
>>> pt1=eval(repr(pt))
>>> pt1
Point2D(1, 2, 'A')
```

Les méthodes __repr__ et __str__

La méthode __str__

- Appelée par la fonction str
- Représentation textuelle informelle (« humaine ») d'un objet
- C'est la méthode qui est exécutée lorsqu'on affiche l'objet ou que l'on crée une chaîne formatée

Classe Point2D du module point.py

```
1    def __str__(self):
2        prefixe = self.id
3        if not prefixe:
4            prefixe = ""
5        return prefixe + f"({self.x},{self.y})"
```

```
>>> f"{pt}"
'A(1, 2)'
```

- Si __repr__ est définie mais non __str__ alors la représentation informelle utilise __repr__ (inverse non vrai)

Opérateurs (fonctions) binaires 1 / 4

Utilisation des opérateurs +, -, etc.

- Il est possible de créer des classes dont les instances pourront être utilisées comme opérandes des opérateurs « arithmétiques », par exemple des classes : Vecteur, Matrice, Fonction (mathématique), etc.

```
>>> from vecteur import Vecteur2D
>>> u = Vecteur2D(1,2,'u')
>>> v = Vecteur2D(2,3,'v')
>>> u+v
Vecteur2D(3, 5, 'u+v')
>>> u-v
Vecteur2D(-1, -1, 'u-v')
```

INSA ROUEN NORMANDIE

Opérateurs (fonctions) binaires 2 / 4

Méthodes spéciales `__add__`, `__sub__`, etc.

- Il faut pour cela définir les méthodes spéciales :
  - `__add__(self, autre)` pour l'opérateur `+`
  - `__sub__(self, autre)` pour l'opérateur `-`
  - `__mul__(self, autre)` pour l'opérateur `*`
  - `__matmul__(self, autre)` pour l'opérateur `@`
  - `__pow__(self, autre[, modulo])` pour l'opérateur `**` ou la fonction `pow`
  - `__truediv__(self, autre)` pour l'opérateur `/`
  - `__floordiv__(self, autre)` pour l'opérateur `//`
  - `__mod__(self, autre)` pour l'opérateur `%`
  - `__divmod__(self, autre)` pour la fonction `divmod`
  - `__lshift__(self, autre)` pour l'opérateur `<<`
  - `__rshift__(self, autre)` pour l'opérateur `>>`
  - `__and__(self, autre)` pour l'opérateur `and`
  - `__or__(self, autre)` pour l'opérateur `or`
  - `__xor__(self, autre)` pour l'opérateur `xor`
- Pour interpréter `x + y`, Python invoque `x.__add__(y)`

- Si le type de l'argument (l'opérande droite de l'opérateur) n'est pas compatible avec l'objet, la valeur retournée doit être `NotImplemented`

Opérateurs (fonctions) binaires 3 / 4

Classe Vecteur2D du module vecteur.py

```
5  class Vecteur2D:
6      def __init__(self, x: float, y:float, identifiant: str):
7          self._x = x
8          self._y = y
9          self._id = identifiant
10
11     @property
12     def x(self) -> float:
13         return self._x
14
15     @property
16     def y(self) -> float:
17         return self._y
18
19     @property
20     def id(self) -> str:
21         return self._id
```

```
38     def __add__(self, vecteur_2d: Vecteur2D) -> Vecteur2D:
39         if not isinstance(vecteur_2d, Vecteur2D):
40             return NotImplemented
41         return Vecteur2D(self.x + vecteur_2d.x,
42                          self.y + vecteur_2d.y,
43                          self.id + "+" + vecteur_2d.id)
44
45     def __sub__(self, vecteur_2d: Vecteur2D) -> Vecteur2D:
46         if not isinstance(vecteur_2d, Vecteur2D):
47             return NotImplemented
48         return Vecteur2D(self.x - vecteur_2d.x,
49                          self.y - vecteur_2d.y,
50                          self.id + "-" + vecteur_2d.id)
51
52     def __mul__(self, vecteur_2d: Vecteur2D) -> float:
53         if not isinstance(vecteur_2d, Vecteur2D):
54             return NotImplemented
55         return self.x*vecteur_2d.x + self.y*vecteur_2d.y
```

INSA ROUEN NORMANDIE

Opérateurs (fonctions) binaires 4 / 4

Méthodes spéciales réflexives : `__radd__`, `__rsub__`, etc.

- Si `x + y` est utilisée et que la classe de `x` ne possède pas la méthode `__add__` ou qu'elle retourne `NotImplemented`, Python essaye d'interpréter `y.__radd__(x)`
- Si la classe de `y` ne définit la méthode `__radd__(self, other)` ou qu'elle retourne `NotImplemented` alors une exception `TypeError` est levée

Méthodes spéciales pour les opérations arithmétiques augmentées : `__iadd__`, `__isub__`, etc.

- Utilisées pour les opérateurs arithmétiques associés à l'affectation : l'objet doit donc être mutable
- Si `x += y` (équivalent à `x = x + y`) est utilisée alors Python interprète `x.__iadd__(y)`

INSA ROUEN NORMANDIE

Opérateurs (fonctions) unaires

- Pour utiliser les opérateurs unaires +, -, etc. il faut définir les méthodes spéciales :
  - `__pos__(self)` pour l'opérateur +
  - `__neg__(self)` pour l'opérateur -
  - `__abs__(self)` pour la fonction abs
  - `__invert__(self)` pour l'opérateur ~

Classe `Point2D` du module `point.py`

```
43    def __abs__(self):
44        return Point2D(abs(self.x), abs(self.y))
```

```
>>> abs(Point2D(-3,2))
Point2D(3,2)
```

Fonctions de transtypage numériques

int, float, etc.

- L'utilisation d'une fonction de transtypage numérique appelle une méthode spéciale
  - `__int__(self)` pour la fonction `int`
  - `__round__(self)` pour la fonction `round`
  - `__float__(self)` pour la fonction `float`
  - `__complex__(self)` pour la fonction `complex`

Classe Point2D du module point.py

```
46    def __complex__(self):
47        return complex(self.x, self.y)
```

```
>>> complex(Point2D(-3,2))
(-3+2j)
```

Conclusion

Nous avons vu dans ce cours

- Le rôle des méthodes spéciales
- Deux représentations des objets sous forme de chaine de caractères
- Comment permettre l'utilisation des opérateurs et fonctions usuels

Rappel : principe général de python

Le concepteur/développeur d'une API écrit plus de code pour simplifier la vie de l'utilisateur

INSA ROUEN NORMANDIE

Python
Méthodes spéciales (suite)

Nicolas Delestre

INSA
ROUEN NORMANDIE

Identité et égalité

Rappels

- Les variables Python référencent des objets : on obtient la référence d'un objet grâce à la fonction standard `id`
- Deux variables peuvent référencer le même objet (utilisation de l'affectation) : on compare les références des objets grâce à l'opérateur `is`
- Deux objets peuvent être égaux : les attributs les caractérisant sont égaux
- L'opérateur == permet de tester l'égalité. Par défaut pour Python, cette égalité repose sur l'identité
- Pour les types de base, ce comportement a été redéfini

Illustration

```
>>> class A:
        pass
>>> a1=A()
>>> a2=A()
>>> a3=a1
```

```
>>> a1 is a2
False
>>> a1 is a3
True
>>> a1==a2
False
>>> a1==a3
True
```

```
>>> x=1.0
>>> y=1.0
>>> x is y
False
>>> x==y
True
```

Attention aux int

```
>>> a=256
>>> b=256
>>> a is b
True
>>> a=257
>>> b=257
>>> a is b
False
```

Les méthodes spéciales __eq__ et __ne__ 1 / 2

La méthode __eq__

- C'est la méthode qui est exécutée lorsque l'opérateur == est utilisé
  Ainsi l'interprétation `a == b` invoque `a.__eq__(b)`
- Code de `__eq__(self, autre)` (bonne pratique) :
  - Tester si le paramètre est un instance de la classe de `self` :
    `isinstance(autre, LaClasseDeSelf)` ou `isinstance(autre, self.__class__)`
  - Tester les égalités des attributs qui définissent l'égalité
  - Renvoyer `NotImplemented` si l'opérateur d'égalité n'a pas à être utilisé

La méthode __ne__

- C'est la méthode qui est exécutée lorsque l'opérateur `!=` est utilisé
- Par défaut `__ne__` renvoie la négation de l'appel à `__eq__` (sauf si le résultat est `NotImplemented`)

Les méthodes spéciales __eq__ et __ne__ 2 / 2

Point2D

```
def __eq__(self, autre):
    if not isinstance(autre, self.__class__):
        return False
    else:
        return self.x == autre.x and self.y == autre.y
```

Point3D

```
def __eq__(self, autre):
    if not isinstance(autre, self.__class__):
        return False
    else:
        return super().__eq__(autre) and self.z == autre.z
```

ROUEN NORMANDIE

Les méthodes spéciales de comparaison d'ordre

`__ge__`, `__gt__`, ...

- Les méthodes spéciales sont :
  - `__ge__` pour l'opérateur $\geq$
  - `__gt__` pour l'opérateur $>$
  - `__le__` pour l'opérateur $\leq$
  - `__lt__` pour l'opérateur $<$
- Le fait de développer `__lt__` (inversement `__gt__`) n'oblige pas à développer `__gt__` (inversement `__lt__`)

Attention

- Il n'y a pas de lien entre `__le__` ou `__ge__` et `__lt__`, `__gt__` et `__eq__` : par exemple le fait de coder uniquement `__lt__` et `__eq__` ne permet d'utiliser l'opérateur `<=`
- Il n'y a pas de lien entre `__le__` et `__ge__`
- Le décorateur `total_ordering` du module standard `funtools` permet de lever ces limites

La méthode spéciale `__hash__` 1 / 2

Rappels

- L'empreinte (*hash*) d'un objet est un nombre (inférieur à une certaine borne) qui le caractérise :
  - Deux objets égaux doivent avoir la même empreinte
  - Deux objets différents devraient avoir deux empreintes différentes (mais il peut y avoir des collisions)

La méthode `__hash__`

- En Python on obtient l'empreinte d'un objet en appelant la fonction standard `hash` qui appelle la méthode spéciale `__hash__`
- Si la méthode `__hash__` n'est pas définie, la fonction `hash` lève une exception `TypeError`
- Si la méthode `__hash__` est définie, on dit que les objets de cette classe sont *hashable*
- Pour qu'un objet puisse être utilisé comme clé d'un `dict`, ou ajouté à un set ou à un `frozenset`, il faut qu'il soit *hashable*

La méthode spéciale `__hash__` 2 / 2

Bonnes pratiques

- Dans le cas ou l'on veut que les objets soient *hashables* :
  - on définit `__hash__` en combinant les empreintes des attributs immuables de l'objet (par exemple en calculant le *hash* des additions des *hash* des attributs immuables)
  - la méthode `__eq__` doit être redéfinie : `x == y` doit impliquer `hash(x) == hash(y)`

INSA ROUEN NORMANDIE

Objet appelable

- Un appelable (*callable*) est un objet qui peut etre utilisé comme une fonction
- Tout objet dont la classe possède la méthode `__call__(self[,arg, ...])` est un appelable

Point2D

```
def __call__(self, x, y):
    self.x = x
    self.y = y
```

```
>>> from point import Point2D
>>> pt=Point2D(1,2)
>>> pt(3,4)
>>> pt.x
3
```

Conclusion

- Rappels : variables, références aux objets, identité, égalité, empreinte
- Les méthodes spéciales pour les comparaisons
- Les méthodes spéciales qui permettent aux instances d'une classe d'être :
  - *hashables*
  - appelables

INSA ROUEN NORMANDIE

Python
Méthodes spéciales (fin)

Nicolas Delestre

INSA
ROUEN NORMANDIE

Rappels

Les conteneurs

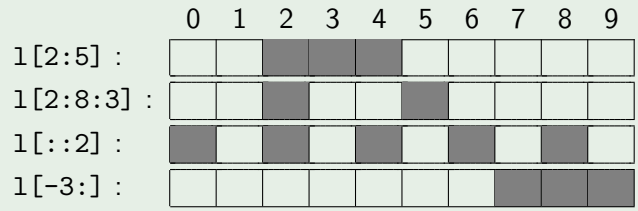
- Définition :
  - « Les conteneurs sont habituellement des séquences (telles que les tuples ou les listes) ou des tableaux de correspondances (comme les dictionnaires)... »[a]
- Un certain nombre d'opérations et de fonctions standards permettent de manipuler les conteneurs :
  - `+`, `in`, `[i]`, `[i:j]`, `[i:j:k]`, etc.
  - `len`, `del`, `reversed`

a. `https://docs.python.org/fr/3/reference/datamodel.html`

INSA ROUEN NORMANDIE

Les *slices*

Les *slices* par l'exemple



Des objets de type `slice`

- Les notations précédentes sont transformées en objet de type `slice`
- Signature de `__init__` : `slice(debut, fin[, pas])`, avec utilisation de la valeur `None`

$$\texttt{l[::2]} \iff \texttt{l[slice(None,None,2)]}$$

- Les objets de type `slice` ont trois attributs en lecture seule : `start`, `stop` et `step`

Méthode spéciales

- `__getitem__(self, cle)` pour l'opérateur `[ ]` (en lecture)
- `__setitem__(self, cle, valeur)` pour l'opérateur `[ ]` (en écriture)
- `__delitem__(self, cle)` pour la suppression (≪ fonction ≫ del) de la valeur associée à la clé

- `__add__(self, autre)` pour l'opérateur `+`
- `__contains__(self, element)` pour l'opérateur `in`
- `__len__(self)` pour la fonction `len`
- `__reversed__(self)` pour la fonction `reversed`

- `__iter__(self)` pour la fonction `iter` ou lorsque le conteneur est utilisé comme itérateur

Un exemple 1 / 5

polyligne.py

```
 5 class Polyligne(object):
 6     def __init__(self, est_fermee, pt1, pt2, *args):
 7         self._est_fermee = est_fermee
 8         self._points = [pt1, pt2]
 9         for pt in args:
10             self.ajouter(pt)
11
12     @property
13     def est_fermee(self):
14         return self._est_fermee
15
16     def ouvrir(self):
17         self._est_fermee = False
18
19     def fermer(self):
20         self._est_fermee = True
21
22     def ajouter(self, *args):
23         for pt in args:
24             self._points.append(pt)
```

Un exemple 2 / 5

polyligne.py

```
26      def __getitem__(self, indice_ou_slice):
27          if isinstance(indice_ou_slice, slice):
28              return Polyligne(self.est_fermee, *self._points[indice_ou_slice])
29          return self._points[indice_ou_slice]
30
31      def __setitem__(self, indice_ou_slice, pt_ou_pts):
32          if isinstance(indice_ou_slice, int):
33              self._points[indice_ou_slice] = pt_ou_pts
34          elif isinstance(indice_ou_slice, slice):
35              if isinstance(pt_ou_pts, Polyligne):
36                  self._points[indice_ou_slice] = pt_ou_pts._points
37              elif isinstance(pt_ou_pts, list) or \
38                   isinstance(pt_ou_pts, tuple):
39                  self._points[indice_ou_slice] = pt_ou_pts
40
41      def __delitem__(self, indice):
42          del(self._points[indice])
```

ROUEN NORMANDIE

Un exemple 3 / 5

polyligne.py

```
44      def __add__(self, autre):
45          return Polyligne(self.est_fermee, *(self._points + autre._points))
46
47      def __len__(self):
48          return len(self._points)
49
50      def __contains__(self, pt):
51          return pt in self._points
52
53      def __iter__(self):
54          return iter(self._points)
55
56      def __repr__(self):
57          return f"Polyligne({self.est_fermee}, {', '.join([repr(pt) for pt in self])})"
58
59      def __str__(self):
60          return f"({', '.join([str(pt) for pt in self])})"
```

ROUEN NORMANDIE

Utilisation de la classe Polyligne

```
>>> from point import Point2D
>>> from polyligne import Polyligne
>>> pl1 = Polyligne(True, Point2D(1,2), Point2D(3,4), Point2D(2,6))
>>> len(pl1)
3
>>> pl1[1:]
Polyligne(True, Point2D(3, 4), Point2D(2, 6))
>>> for pt in pl1:print(pt)
(1, 2)
(3, 4)
(2, 6)
>>> pl2 = Polyligne(True, Point2D(5, 2), Point2D(7, 4), Point2D(8, 6))
>>> pl1[0:2] = pl2
>>> pl1
Polyligne(True, Point2D(5, 2), Point2D(7, 4), Point2D(8, 6),
Point2D(2, 6))
```

INSA ROUEN NORMANDIE

Utilisation de la classe Polyligne

```
>>> l = [Point2D(9,4), Point2D(10,8)]
>>> pl1[-1:]=l
>>> pl1
Polyligne(True, Point2D(5, 2), Point2D(7, 4), Point2D(8, 6),
Point2D(9, 4), Point2D(10, 8))
>>> del(pl1[2:])
>>> pl1
Polyligne(True, Point2D(5, 2), Point2D(7, 4))
>>> Point2D(5,2) in pl1
True
```

INSA ROUEN NORMANDIE

Conclusion

- Rappels sur la notion de conteneur et les *slices*
- Les opérateurs et fonctions standards sur les conteneurs font appel à des méthodes spéciales

INSA ROUEN NORMANDIE

Python
Attributs et méthodes de classe
Méthodes statiques

Nicolas Delestre

INSA
ROUEN NORMANDIE

Attributs de classe

Déclaration, utilisation

- On déclare un attribut de classe en dehors de toute méthode, en ne le préfixant pas par `self` :
  - il est souvent initialisé juste après la déclaration de la classe
  - il est utilisé dans les méthodes de classe ou les méthodes d'objet

point.py

```
 5 class Point2D:
 6     _les_points = []

16     def __init__(self, x: float, y: float, identifiant: str=None):
17         self._x = x
18         self._y = y
19         self._id = identifiant
20         self._les_points.append(self)
```

Méthodes de classe 1 / 2

Déclaration

- On déclare une méthode de classe :
  - en préfixant la déclaration du décorateur `@classmethod`
  - en utilisant uniquement des attributs de classe
  - en remplaçant le `self` par `cls` (bonne pratique)

point.py

```
 5 class Point2D:
 6     _les_points = []
 7
 8     @classmethod
 9     def les_points(cls):
10         return cls._les_points
```

INSA ROUEN NORMANDIE

Méthodes de classe 2 / 2

Utilisation

- On utilise une méthode de classe en préfixant l'appel de la méthode par le nom de classe ou par un objet de la classe (notation pointée)

```
>>> from point import Point2D
>>> Point2D.les_points()
[]
>>> pt1=Point2D(1,1)
>>> Point2D(2,2)
Point2D(2, 2)
>>> Point2D.les_points()
[Point2D(1, 1), Point2D(2, 2)]
>>> pt1.les_points()
[Point2D(1, 1), Point2D(2, 2)]
```

INSA ROUEN NORMANDIE

Méthodes statiques 1 / 2

Déclaration

- On déclare une méthode statique en préfixant la déclaration du décorateur `@staticmethod`
- Une méthode statique n'a pas accès à l'état de la classe, il s'agit d'une fonction utilitaire rattachée à la classe (attention aux faux amis avec d'autres langages tels que le Java)

point.py

```
12    @staticmethod
13    def translation(pt, vecteur, id=None):
14        return Point2D(pt.x + vecteur.x, pt.y + vecteur.y, id)
```

INSA ROUEN NORMANDIE

Utilisation

- On utilise une méthode statique en préfixant l'appel de la méthode par le nom de classe ou un objet de la classe (notation pointée)

```
1 >>> from point import Point2D
2 >>> from vecteur import Vecteur2D
3 >>> Point2D.translation(Point2D(1,1), Vecteur2D(1,0,'v'))
4 Point2D(2, 1)
```

INSA ROUEN NORMANDIE

Conclusion

- Attributs et méthodes de classes
- Méthodes statiques
- Utilisable depuis la classe ou depuis une instance de la classe

INSA ROUEN NORMANDIE

Python
Les énumérations

Nicolas Delestre

INSA
ROUEN NORMANDIE

Rappels

Définitions

- Définition générale : ≪ Un type énuméré (appelé souvent énumération ou juste enum, parfois type énumératif ou liste énumérative) est un type de données qui consiste en un ensemble de valeurs constantes ≫ (Wikipédia)

- Dans la cadre de Python : ≪ Une énumération est un ensemble de noms symboliques, appelés membres, liés à des valeurs constantes et uniques. Au sein d'une énumération, les membres peuvent être comparés entre eux et il est possible d'itérer sur l'énumération elle-même. ≫ (Documentation Python)

INSA ROUEN NORMANDIE

Création d'une énumération

- La création d'une énumération est réalisée par la création d'une classe possédant autant d'attributs de classe qu'il y a d'éléments dans l'énumération
- On crée cette classe énumération à partir de la classe `Enum` qui est proposée par le module standard `enum`
- Deux méthodes de création :
  1. par héritage de la classe `Enum`
  2. par instanciation de la classe `Enum` (qui retourne une classe)

INSA ROUEN NORMANDIE

Création par héritage 1 / 2

- Création d'une sous classe de la classe `Enum` avec déclaration d'un attribut de classe pour chaque élément de l'énumération
  - la bonne pratique veut que les identifiants des attributs soient en majuscule
  - on peut ajouter/redéfinir des méthodes
  - l'héritage d'une énumération est autorisée si la super classe (sous classe d'`Enum`) n'a défini aucun attribut de classe
- À chaque élément de l'énumération est associé une valeur, qui peut être de tout type (`int`, `str`, etc.). Cette valeur peut être attribuée automatiquement en utilisant des instances de la classe `auto` (à partir de python 3.6)

```
class Couleur(Enum):
    BLANC = 1
    NOIR = 2
    BLEU = 3
    ROUGE = 4
    VERT = 5
```

```
class Jour(Enum):
    LUNDI = auto()
    MARDI = auto()
    MERCREDI = auto()
    JEUDI = auto()
    VENDREDI = auto()
    SAMEDI = auto()
    DIMANCHE = auto()
```

Création par héritage 2 / 2

- Deux éléments (attributs de classe) peuvent avoir la même valeur (sauf si utilisation du décorateur de classe `unique`) : le deuxième est dit *alias* du premier

```
class JourEntreprise(Enum):
    OUVRE = 1
    FERIE= 2
    DIMANCHE = 2
```

```
>>> from enum import unique
>>> @unique
... class ERREUR(Enum):
...     VAL1 = 1
...     VAL2 = 2
...     ALIAS_VAL2 = 2
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
  ...
    raise ValueError('duplicate values found in %r: %s' %
ValueError: duplicate values found in <enum 'ERREUR'>: ALIAS_VAL2 -> VAL2
```

INSA ROUEN NORMANDIE

Création par instanciation de la classe Enum

- L'instanciation de la classe `Enum` retourne une classe énumération
- L'initialiseur admet comme paramètre formel (deux premiers obligatoires) :
  - `value` : le nom de la classe énumération à créer (la bonne pratique veut que ce nom soit identique à l'identifiant de la variable qui va référencer la classe)
  - `names` : une chaîne de caractères (éléments séparés par une virgule et/ou des espaces), ou une séquence listant le nom des éléments, ou un dictionnaire
  - `module` : le nom du module de la classe créée
  - `qualname` : la position de la classe dans le module
  - `type` : type des valeurs des éléments
  - `start` : valeur du premier élément (à partir de Python 3.5)
- Par défaut, les valeurs sont des `int` et la valeur du premier élément vaut 1
- Inconvénient : il est impossible d'ajouter ou de redéfinir des méthodes

```
JourDeLaSemaine = Enum("JourDeLaSemaine", "LUNDI MARDI MERCREDI JEUDI VENDREDI SAMEDI DIMANCHE")
Couleur = Enum("Couleur", {"BLANC":"A", "NOIR":"B", "BLEU":"C", "ROUGE":"D", "VERT":"F"})
```

Utilisation d'une énumération

- On peut utiliser une classe énumération :
  - en utilisant l'attribut de classe
  - en « instanciant » la classe
  - en utilisant la classe comme un dictionnaire (les clés sont les identifiants des attributs de classe en chaîne de caractères)
  - en utilisant la classe comme une séquence (indexée par le nom des valeurs)
- Les instances ont deux attributs : `value` et `name`

```
>>> from couleur import Couleur
>>> Couleur.BLANC
<Couleur.BLANC: 1>
>>> Couleur(1)
<Couleur.BLANC: 1>
>>> Couleur['BLANC']
<Couleur.BLANC: 1>
```

```
>>> c1=Couleur(1)
>>> c2=Couleur(1)
>>> c1 is c2
True
```

```
>>> for c in Couleur:
...     print(c)
...
Couleur.BLANC
Couleur.NOIR
Couleur.BLEU
Couleur.ROUGE
Couleur.VERT
```

```
>>> c=Couleur(1)
>>> c.value
1
>>> c.name
'BLANC'
```

Autres classes proposées par le module enum 1 / 3

IntEnum

≪ Classe de base pour créer une énumération de constantes qui sont également des sous-classes de int ≫ (documentation Python)

Extrait de https://stackoverflow.com/questions/52929954/difference-between-enum-and-intenum-in-python

```
>>> class Shape(IntEnum):
...     CIRCLE = 1
...     SQUARE = 2
...
>>> class Color(Enum):
...     RED = 1
...     GREEN = 2
...
```

```
>>> Shape.CIRCLE == Color.RED
False
>>> Shape.CIRCLE == 1
True
>>> Shape.CIRCLE+Shape.SQUARE
3
>>> Color.RED+Color.GREEN
--------------------------------------------------------------------------
TypeError
...
TypeError: unsupported operand type(s) for +: 'Color' and 'Color'
```

Flag

≪ Classe de base pour créer une énumération de constantes pouvant être combinées avec des opérateurs de comparaison bit-à-bit, sans perdre leur qualité de Flag ≫ (à partir de Python 3.6, documentation Python)

Extrait de http://zetcode.com/python/enum/

```
>>> class Perm(Flag):
...     EXECUTE = auto()
...     WRITE = auto()
...     READ = auto()
...
```

```
>>> list(Perm)
[<Perm.EXECUTE: 1>, <Perm.WRITE: 2>, <Perm.READ: 4>]
>>> Perm.EXECUTE | Perm.READ
<Perm.READ|EXECUTE: 5>
>>> Perm.EXECUTE & Perm.READ
<Perm.0: 0>
```

ROUEN NORMANDIE

Autres classes proposées par le module enum 3 / 3

IntFlag

≪ Classe de base pour créer une énumération de constantes pouvant être combinées avec des opérateurs de comparaison bit-à-bit, sans perdre leur qualité de IntFlag. Les membres de IntFlag sont aussi des sous-classes de int. ≫ (à partir de Python 3.6, documentation Python)

Extrait de http://zetcode.com/python/enum/

```
>>> class Perm(IntFlag):
...     EXECUTE = auto()
...     WRITE = auto()
...     READ = auto()
...
```

```
>>> list(Perm)
[<Perm.EXECUTE: 1>, <Perm.WRITE: 2>, <Perm.READ: 4>]
>>> Perm.EXECUTE | Perm.READ
<Perm.READ|EXECUTE: 5>
>>> Perm.EXECUTE + Perm.READ
5
```

ROUEN NORMANDIE

Conclusion

Nous avons vu :

- des rappels sur les énumérations
- qu'est ce qu'une énumération en Python
- les deux méthodes pour créer une énumération : par héritage ou par instanciation de la classe `Enum`
- comment utiliser une énumération
- trois autres types d'énumération à étendre proposés par `enum` : `IntEnum`, `Flag` et `FlagEnum`

INSA ROUEN NORMANDIE

Python
Les exceptions

Nicolas Delestre

INSA
ROUEN NORMANDIE

Rappels 1 / 3

Qu'est ce qu'une exception ?

- C'est un mécanisme de gestion des erreurs :
  - Certaines utilisations d'instructions, d'opérations, de fonctions, de méthodes sont susceptibles de générer des erreurs (erreurs systèmes, valeurs incompatibles, types incompatibles, etc.)
  - Plutôt que de mélanger le code de gestion des erreurs avec le code métier, on sépare les deux pour rendre le code plus lisible
  - Plutôt que déterminer le type d'erreur en fonction d'une valeur particulière, ce sont des classes (une exception est donc un objet) qui vont indiquer la sémantique de l'erreur. Les langages proposent une hiérarchie d'exceptions, que le programmeur peut étendre.
- Une exception est dite levée lorsqu'une erreur apparaît
- Une exception est dite capturée lorsqu'elle est est gérée et traitée. Ce cas peut lever d'autres exceptions

Rappels 2 / 3

Comment gérer une exception ?

- Lorsqu'on utilise une suite d'instructions qui est susceptible de lever une exception :
  - soit on capture l'exception et on exécute une suite d'instructions qui a pour objectif de gérer l'erreur
  - soit on ne capture pas l'exception et dans ce cas, la fonction ou méthode en cours s'arrête et propage l'exception levée à la fonction appelante. Si cette propagation remonte jusqu'au programme principal sans que l'exception soit capturée, le programme s'arrête

INSA ROUEN NORMANDIE

Rappels 3 / 3

Quand lever une exception ?

Deux cas :

1. Une erreur « système » est détectée. Par exemple
   - ouverture d'un fichier qui n'existe pas
   - connexion réseau qui n'a pu être établie
   - etc.
2. L'appelant de la fonction ne respecte pas les règles définies par sa « documentation »[a]. Le corps de la fonction commence donc par une suite de tests susceptible de lever une exception.
   - Il y a un débat sur la vérification des types. Python adopte le principe du *duck typing*, il ne devrait pas y avoir de vérification de type. Mais on voit apparaître différents packages de vérification de type à partir des annotations (par exemple typecheck : `https://github.com/prechelt/typecheck-decorator`).

a. les langages compilés à typage statique vérifient une partie de cette documentation en vérifiant l'adéquation des types. Plus les types sont bien définis plus cette vérification à la compilation est utile

Les exceptions standards

https://docs.python.org/3/library/exceptions.html

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StopAsyncIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EOFError
      +-- ImportError
           +-- ModuleNotFoundError
      +-- LookupError
      |    +-- IndexError
      |    +-- KeyError
      +-- MemoryError
      +-- NameError
      |    +-- UnboundLocalError
```

```
BaseException
 +-- Exception
      +-- OSError
      |    +-- BlockingIOError
      |    +-- ChildProcessError
      |    +-- ConnectionError
      |    |    +-- BrokenPipeError
      |    |    +-- ConnectionAbortedError
      |    |    +-- ConnectionRefusedError
      |    |    +-- ConnectionResetError
      |    +-- FileExistsError
      |    +-- FileNotFoundError
      |    +-- InterruptedError
      |    +-- IsADirectoryError
      |    +-- NotADirectoryError
      |    +-- PermissionError
      |    +-- ProcessLookupError
      |    +-- TimeoutError
      +-- ReferenceError
      +-- RuntimeError
      |    +-- NotImplementedError
      |    +-- RecursionError
      +-- SyntaxError
      |    +-- IndentationError
      |         +-- TabError
      +-- SystemError
      +-- TypeError
```

```
BaseException
 +-- Exception
      +-- ValueError
      |    +-- UnicodeError
      |         +-- UnicodeDecodeError
      |         +-- UnicodeEncodeError
      |         +-- UnicodeTranslateError
      +-- Warning
           +-- DeprecationWarning
           +-- PendingDeprecationWarning
           +-- RuntimeWarning
           +-- SyntaxWarning
           +-- UserWarning
           +-- FutureWarning
           +-- ImportWarning
           +-- UnicodeWarning
           +-- BytesWarning
           +-- ResourceWarning
```

Lever d'une exception

raise

- On lève une exception à l'aide l'instruction `raise` suivie de la création d'une instance d'une sous classe de `Exception`
- Le constructeur prend au moins en paramètre une chaîne de caractères explicitant l'erreur

Exemple : simuler une classe abstraite

```
class ClasseAbstraite:
    def m1(self):
        raise NotImplementedError("Classe abstraite")

    def m2(self):
        self.m1()

class ClasseConcrete(ClasseAbstraite):
    def m1(self):
        pass
```

Attraper une exception

try...except

- La suite d'instructions susceptible de lever une exception, que l'on veut capturer, est :
  - précédée de l'instruction `try:`
  - suivie d'au moins une clause `except` suivie de la classe (ou d'un tuple de classes) de l'exception devant être attrapée avec possibilité de récupérer l'exception (mot clé `as`)
  - suivie optionnellement d'une clause `else`. Cette clause doit être positionnée après toutes les clauses `except` (il ne peut pas y avoir de clause `except` après un `else`), son code est exécuté lorsqu'aucune exception a été levée
  - suivie optionnellement d'une clause `finally`. Cette clause doit être positionnée après toutes les clauses `except` et la clause `else`, son code est toujours exécuté (qu'il y ait eu une exception ou pas).

```
...
try:
    instr1
    ...
except Exception1:
    instr1
except (Exception2, \
        Exception2) as ex:
    ...
else:
    ...
finally:
    ...
...
```

Création d'un type d'exception

Exception

- Pour créer un type d'exception il faut créer une sous classe (ou sous-sous classe) de la classe `Exception` (importée par défaut en python 3)
- On peut la configurer comme toute classe (redéfinir `__init__`, ajouter des attributs, ajouter des méthodes) mais elles sont souvent très simples (voire vides, utilisation de `pass`)
- La bonne pratique veut que le nom d'une exception se termine par `Error` (ou `Erreur` si codage en français).

INSA ROUEN NORMANDIE

Exemple : polyligne.py 1 / 6

Règles

- Un polyligne n'a de sens que s'il y a au moins deux points
- On interdit qu'une polyligne possède deux fois le même point

Définition des exceptions

```
4 class MemePointInterditErreur(Exception):
5     pass
6
7 class AuMoinsDeuxPointsErreur(Exception):
8     pass
```

INSA ROUEN NORMANDIE

Exemple : polyligne.py 2 / 6

Une méthode statique pour vérifier l'unicité des points (au sens de l'égalité)

```
12      @staticmethod
13      def _verifier_points_differents_deux_a_deux(pts):
14          if pts:
15              if pts[0] in pts[1:]:
16                  raise MemePointInterditErreur(f"{pts[0]} est présent au moins deux fois")
17              return Polyligne._verifier_points_differents_deux_a_deux(pts[1:])
```

Une méthode statique pour vérifier le nombre de points minimal

```
19      @staticmethod
20      def _verifier_assez_points(pts):
21          if len(pts) < 2:
22              raise AuMoinsDeuxPointsErreur("Une polyligne doit avoir au moins deux points")
```

Exemple : polyligne.py 3 / 6

Redéfinition de __init__

```
24    def __init__(self, est_fermee, pt1, pt2, *args):
25        pts = [pt1, pt2] + list(args)
26        self._verifier_points_differents_deux_a_deux(pts)
27        self._est_fermee = est_fermee
28        self._points = pts
```

Redéfinition de ajouter

```
40    def ajouter(self, *args):
41        pts = self._points + list(args)
42        self._verifier_points_differents_deux_a_deux(pts)
43        self._points = pts
```

INSA ROUEN NORMANDIE

Exemple : polyligne.py 4 / 6

Redéfinition de __setitem__

```
50    def __setitem__(self, indice_ou_slice, pt_ou_pts):
51        pts = list(self._points)
52        if isinstance(indice_ou_slice, int):
53            pts[indice_ou_slice] = pt_ou_pts
54        elif isinstance(indice_ou_slice, slice):
55            if isinstance(pt_ou_pts, Polyligne):
56                pts[indice_ou_slice] = pt_ou_pts._points
57            elif isinstance(pt_ou_pts, list) or \
58                 isinstance(pt_ou_pts, tuple):
59                pts[indice_ou_slice] = pt_ou_pts
60
61        self._verifier_points_differents_deux_a_deux(pts)
62        self._verifier_assez_points(pts)
63
64        self._points = pts
```

Exemple : polyligne.py 5 / 6

Redéfinition de __delitem__

```
66      def __delitem__(self, indice_ou_slice):
67          pts = list(self._points)
68          del(pts[indice_ou_slice])
69          self._verifier_assez_points(pts)
70          self._points = pts
```

Exemple : polyligne.py 6 / 6

```
>>> from point import Point2D
>>> from polyligne import Polyligne
>>> pl=Polyligne(True, Point2D(1,2), Point2D(3,4), Point2D(2,6))
>>> pl.ajouter(Point2D(1,2))
---------------------------------------------------------------------------
MemePointInterditErreur               Traceback (most recent call last)
<ipython-input-4-019abfa3dcb3> in <module>
----> 1 pl.ajouter(Point2D(1,2))
...
MemePointInterditErreur: (1,2) est présent au moins deux fois

>>> pl[0:2]=[Point2D(0,0), Point2D(1,2)]
>>> pl
Polyligne(True, Point2D(0, 0), Point2D(1, 2), Point2D(2, 6))
>>> pl[0]=Point2D(1,2)
---------------------------------------------------------------------------
MemePointInterditErreur               Traceback (most recent call last)
<ipython-input-10-733cd8ccec92> in <module>
----> 1 pl[0]=Point2D(1,2)
...
MemePointInterditErreur: (1,2) est présent au moins deux fois
>>> del(pl[0:2])
---------------------------------------------------------------------------
AuMoinsDeuxPointsErreur               Traceback (most recent call last)
<ipython-input-12-0ffab3dd44e2> in <module>
----> 1 del(pl[0:2])
...
AuMoinsDeuxPointsErreur: Une polyligne doit avoir au moins deux points
```

Conclusion

- Nous avons rappelé ce que sont les exceptions, ce que signifie lever et attraper une exception
- Nous avons rapidement vu les exceptions standards de Python et comment on lève ou attrape une exception
- Nous avons mis en pratique cela en ajoutant des exceptions à la classe `Polyligne`

INSA ROUEN NORMANDIE

Python
L'instruction `with` et les gestionnaires de contexte

Nicolas Delestre

INSA
ROUEN NORMANDIE

Un motif de développement courant

Exemple de motif courant

```
1 # instructions I1 d'initialisation d'un objet o
2 try :
3     # instructions I2 utilisant l'objet o
4     # pouvant lever une exception
5 except UneException as e:
6     # instructions I3 s'il y a eu une exception
7  finally :
8     # instructions I4 à réaliser qu'il y ait eu exception ou pas
```

Dans quel contexte ?

- l'utilisation d'un fichier (ouverture, lecture/écriture, fermeture)
- communication réseau
- etc.

Un exemple avec la lecture d'un fichier

Exemple : affichage du contenu d'un fichier

```
1 f = open("fichier.txt", "rt")
2 try:
3     for ligne in f:
4         print(ligne, end="")
5 except Exception:
6     pass
7 finally :
8     f.close()
```

L'instruction `with`

with [...as]

- utilisation d'un gestionnaire de contexte (*context manager*) qui :
  - possède et exécute *I1*
  - retourne l'objet *o* afin de pouvoir exécuter la liste des instruction *I2*
  - possède et exécute automatiquement *I3* et *I4* une fois les instructions *I2* entièrement exécutées ou arrêtées par une exception
- Deux syntaxes :

```
gc = gestionnaire_de_contexte()
with gc:
    I2(gc)
```

```
with gestionnaire_de_contexte() [as gc]:
    I2(gc)
```

Inspire de https://www.afpy.org/doc/python/3.5/tutorial/errors.html

```
with open("fichier.txt", "rt") as f:
    for ligne in f:
        print(ligne, end="")
```

À l'aide d'une classe

- Toute classe qui possède les méthodes :
  - `__enter__(self)` qui exécute *I1* retourne l'objet o
  - `__exit__(self, except_type, except_value, traceback)` qui exécute optionnellement *I3* et qui exécute *I4*
    - `except_type` : `None` ou le type de l'exception qui a été levée
    - `except_value` : `None` ou la chaîne de caractères qui décrit l'exception
    - `traceback` : `None` ou un objet permettant d'afficher la trace d'exécution

```
3  class UneException(Exception):
4      pass
5
6  class UnGestionnaireDeContexteJouet:
7      def __enter__(self):
8          print("I1")
9          return #on peut spécifier un objet qui sera récupéré dans le as
11     def __exit__(self, except_type, except_value, traceback):
12         if except_type:
13             print(f"I3 : Type = {except_type}, valeur = {except_value}")
14         print("I4")
```

Exemples d'utilisation

```
>>> with UnGestionnaireDeContexteJouet() as cm:
    print("I2")
I1
I2
I4
>>> with UnGestionnaireDeContexteJouet() as cm:
    print("I2")
    raise Exception("une erreur")
I1
I2
I3 : Type = <class '__main__.UneException'>, valeur = une erreur
I4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File ".../context_manager.py", line 15, in <module>
    raise UneException("une erreur")
UneException: une erreur
```

À l'aide d'une fonction

- Toute fonction décorée par le décorateur `contextmanager` (du module `contextlib`) et qui a le corps suivant :

```
try :
    # instructions à exécuter au début
    yield # optionnellement avec un objet qui sera récupéré dans le as
    # instructions exécutées s'il n'y a pas eu d'exception de levée
except Exception as err:
    # instruction exécutées s'il y a eu une exception de levée
finally :
    # instructions à exécuter après
```

```
16 from contextlib import contextmanager
17
18 @contextmanager
19 def unGestionnaireDeContexteJouet():
20     try :
21         print ("I1")
22         yield #on peut spécifier un objet qui sera récupéré dans le as
23     except Exception as err:
24         print (f"I3 : Type = {type(err)}, valeur = {err}")
25         raise err
26     finally :
27         print ("I4")
```

Exemples d'utilisation

```
>>> with unGestionnaireDeContexteJouet() as cm:
    print("I2")
I1
I2
I4
>>> with unGestionnaireDeContexteJouet() as cm:
    print("I2")
    raise UneException("une erreur")
I1
I2
I3 : Type = <class '__main__.UneException'>, valeur = une erreur
I4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File ".../context_manager.py", line 15, in <module>
    raise UneException("une erreur")
UneException: une erreur
```

Gestionnaires de contexte particuliers

Gestionnaires de contexte réentrants

```
gc = GestionnaireDeContexteReentrant()
with gc:
    # instructions
    with gc:
        # instructions
```

Gestionnaires de contexte réutilisables

```
gc = GestionnaireDeContexteReutilisiable()
with gc:
    # instructions
# instructions
with gc:
    # instructions
```

Conclusion

- Un motif de développement courant
- Utilisation de l'instruction `with`
- Conception de gestionnaire de contexte

INSA ROUEN NORMANDIE

Python
Les Dataclasses

Nicolas Delestre

INSA
ROUEN NORMANDIE

Contexte

La représentation des données

- Très souvent les objets ont besoin de s'échanger des données qui forment un tout mais qui n'ont pas nécessairement des messages à traiter
- Dans les langages tels que C, Pascal, etc. on utilise dans ce cas des **struct**, **record**, etc. pour organiser ces données
- En python, jusqu'à la version 3.7, on utilisait soit :
  - des tuples, mais dans ce cas le code était moins lisible
  - des classes, mais dans ce cas le concepteur devait coder un initialiseur, des transtypages en **str**, etc.

INSA ROUEN NORMANDIE

Module *dataclasses*

- Disponible à partir de la version 3.7 de Python
- Module proposant un décorateur et des fonctions permettant de générer automatiquement du code comme par exemple `__init__` et `__repr__` (cf. la PEP 557) facilitant le développement de classes qui représentent des données

Avantages

- Développement accéléré
- Lisibilité du code

Inconvénient

- Pas d'encapsulation

INSA ROUEN NORMANDIE

Un exemple (tiré de la documentation de python)

Conception

```python
from dataclasses import dataclass

@dataclass
class ElementDInventaire:
    """Classe représentant un élément d'inventaire en stock."""
    nom: str
    prix_unitaire: float
    quantite_en_stock: int = 0

    def cout_total(self) -> float:
        return self.prix_unitaire * self.quantite_en_stock
```

Utilisation

```
>>> from inventaire import ElementDInventaire
>>> ecrous = ElementDInventaire("Ecrou de diametre 15", 0.1,
      100)
>>> ecrous
ElementDInventaire(nom='Ecrou de diametre 15', prix_unitaire
      =0.1, quantite_en_stock=100)
>>> ecrous.nom
'Ecrou de diametre 15'
>>> ecrous.prix_unitaire = 0.2
>>> ecrous.cout_total()
20.0
```

Principaux paramètres du décorateur `dataclass`

À partir de la 3.7

- `init` (valeur par défaut `True`) pour générer la méthode `__init__`
- `repr` (valeur par défaut `True`) pour générer la méthode `__repr__`
- `eq` (valeur par défaut `True`) pour générer la méthode `__eq__`
- `frozen` (valeur par défaut `False`) pour définir les objets immuables
- `unsafe_hash` (valeur par défaut `False`) pour générer la méthode `__hash__` (à n'utiliser que si les objets sont immuables)
- `order` (valeur par défaut `False`) pour générer les méthodes de comparaison (`__lt__`, `__le__`, etc.) en considérant les objets comme des tuples

À partir de la 3.10

- `kw_only` (valeur par défaut `False`) pour obliger à utiliser une initialisation des objets à l'aide d'un passage de paramètre nommé

Quelques fonctions du module dataclasses

```
>>> import dataclasses
>>> dataclasses.asdict(ecrous)
{'nom': 'Ecrou de diametre 15', 'prix_unitaire': 0.1, 'quantite_en_stock':
    100}
>>> dataclasses.astuple(ecrous)
('Ecrou de diametre 15', 0.1, 100)
>>> dataclasses.is_dataclass(ecrous)
True
```

INSA ROUEN NORMANDIE

Fonction field

- Elle permet de paramétrer les valeurs par défaut et le comportement des champs

Exemple tiré de https://www.youtube.com/watch?v=CvQ7e6yUtnw

```python
#!/usr/bin/env python3
import random
import string
from dataclasses import dataclass, field

def id_aleatoire() -> str:
    return "".join(random.choices(string.ascii_uppercase, k=12))

@dataclass
class Personne:
    """Classe représentant une personne"""
    nom: str
    prenom: str
    majeur: bool = True
    emails: list[str] = field(default_factory=list)
    identifiant: str = field(init=False, default_factory=id_aleatoire)
    _champ_recherche: str = field(init=False, repr=False)

    def __post_init__(self):
        self._champ_recherche = f"{self.nom} {self.prenom}"
```

```
>>> from personne import Personne
>>> p = Personne("Delestre","Nicolas")
>>> p
Personne(nom='Delestre', prenom='Nicolas', majeur=True,
      emails=[], identifiant='YZOYLUVTDRSA')
>>>
```

Conclusion

- Les dataclasses en Python, introduites à partir de la version 3.7, simplifient considérablement la création de classes destinées principalement à stocker des données
- Grâce au décorateur `@dataclass`, les méthodes spéciales telles que `__init__`, `__repr__`, et `__eq__` peuvent être générées automatiquement, réduisant ainsi la quantité de code à écrire
- Cela conduit à un développement plus rapide, un code plus lisible, et une meilleure gestion des classes utilisées pour représenter des données
- Le module `dataclasses` offre des fonctionnalités telles que `asdict`, `astuple`, et `field` pour une personnalisation accrue
- Il est essentiel de comprendre les paramètres du décorateur `@dataclass` et la fonction `field` pour tirer le meilleur parti de cette fonctionnalité

INSA ROUEN NORMANDIE

Python
Les annotations (ou *Type hints*)

Nicolas Delestre

INSA
ROUEN NORMANDIE

Typage statique, typage dynamique

- Typage statique : un paramètre formel ou une variable (pour une portée donnée) est associé à un type. Le compilateur peut vérifier si les paramètres effectifs ou les affectations, respectent cette déclaration
- Typage dynamique : le type d'un paramètre formel ou d'une variable, n'est pas associé à un type, son type peut varier au cours du temps

Le Python adopte de typage dynamique

INSA ROUEN NORMANDIE

Documentation formelle et informelle

- Documentation : information sur le code non prise en compte par le compilateur ou l'interpréteur
- Dans la plupart des langages, la documentation est présente dans des commentaires en suivant une certaine syntaxe (javadoc, doxygen, robotdoc, etc.)

Ne pas confondre documentation et commentaire

Cas de Python

- Documentation informelle : les docstring (cf. PEP 257)
- Documentation formelle : les annotations (cf. PEP 484)

INSA ROUEN NORMANDIE

Annotation

- Disponible depuis la version 3.5
- Documentation formelle associée à une entité (paramètre formelle, variable, valeur retournée par une fonction, etc.) qui spécifie le type de cette entité
- Elle peut être utilisée par :
  - l'IDE pour aider le développeur lors du développement
  - des programmes de vérification comme pyright et mypy
- Elle n'est pas utilisée par l'interpréteur Python

Exemple

```
1 def hello_world(ch: str) -> str:
2     return "Bonjour " + ch
```

Collections de base

- Il est possible de spécifier le type des objets que doit contenir une collection (**tuple**, **list** , **set**, **dict**)
  - avant la version 3.9, utilisation des déclarations `Tuple`, `List`, `Set`, `Dict` du module `typing` (*deprecated*)
  - après la version 3.9 directement les types **tuple**, **list** , **set**, **dict**

Exemple

```
1 def somme_des_nombres_pairs(nombres: list[int]) -> int:
2     return sum([nombre for nombre in nombres if nombre % 2 == 0])
```

INSA ROUEN NORMANDIE

Union et Optional

- `Union` (ou, à partir de la version 3.10, l'opérateur barre `|`) permet de déclarer plusieurs types possibles pour une annotation

- `Optional` permet d'indiquer que le type de l'entité peut être aussi `None`

Exemple

```
1 from typing import Union, Optional
2
3 def somme(data: Union[list[int], np.ndarray]) -> Optional[int]:
4     If isinstance(data, list):
5         return sum(data)
6     if isinstance(data, np.ndarray):
7         return int(np.sum(data))
8     return None
9
10 def somme(data: list[int]|np.ndarray]) -> Optional[int]:
11     ...
```

Self

- Lorsque l'on déclare une méthode de classe qui prend en paramètre ou qui retourne un objet du type de la classe que l'on est en train de définir, on ne peut pas utiliser son identifiant :
  - Avant Python 3.10, on utilisait la chaîne de caractères identique à l'identifiant de la classe
  - À partir de Python 3.10, on utilise `Self` du module `typing`

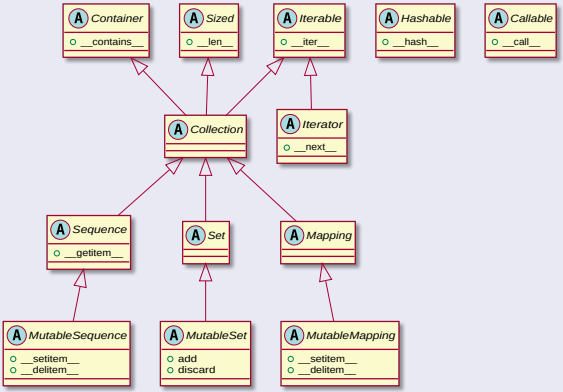
Exemple (extrait de la documentation de Python)

```
1  class Foo:
2      def return_self(self) -> "Foo":
3          ...
4          return self
5
6  from typing import Self
7  class Foo:
8      def return_self(self) -> Self:
9          ...
10         return self
```

Des types génériques

- Plutôt que de spécifier que le paramètre formel d'une fonction doit être un **tuple**, une **list** , un **set**, etc. on peut aussi spécifier que l'on a juste besoin d'itérer sur ce paramètre et donc qu'il est du type `Iterable`

Des types génériques proposés par le module `typing` (issus de `collection.abc`)



Les alias

- Un alias de type permet d'alléger et donner du sens à des annotations complexes
- Créé à l'aide de l'instruction **type** (à partir de 3.12)
- Un alias est une instance de la classe `TypeAliasType`

Exemple (extrait de la documentation de Python)

Plutôt que d'avoir :

```
def broadcast_message(
        message: str,
        servers: Sequence[tuple[tuple[str, int], dict[str, str]]]) -> None:
    ...
```

On utilise les alias :

```
type ConnectionOptions = dict[str, str]
type Address = tuple[str, int]
type Server = tuple[Address, ConnectionOptions]

def broadcast_message(message: str, servers: Sequence[Server]) -> None:
```

La généricité

- Lorsqu'une classe stocke des éléments de même type, il est possible de paramétrer cette classe

```
1 class File[T]:
2     def __init__(self, *args: T):
3         self._elements = list(args)
4
5     def defiler(self) -> T:
6         return self._elements.pop(0)
7
8     def enfiler(self, element: T) -> None:
9         self._elements.append(element)
10 ...
```

- Avant la version 3.11, il fallait déclarer `T` :

```
T = TypeVar('T')
```

Conclusion

- À partir de la version 3.5, Python propose d'annoter classes, paramètres formels, variable et valeur de retour de fonction
- Ces informations formelles sont utilisées par les IDE et des programmes de vérification de type statique (par exemple pyright et mypy)
- Ces informations ne sont pas utilisées par l'interpréteur Python
- L'utilisation des annotations permet d'améliorer la qualité du code

INSA ROUEN NORMANDIE

Python
Les classes abstraites et les protocoles

Nicolas Delestre

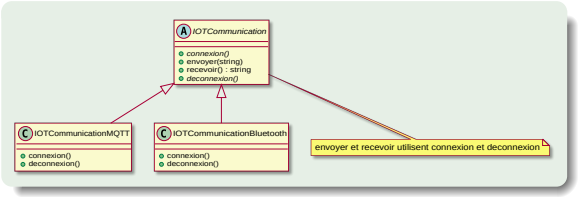
INSA
ROUEN NORMANDIE

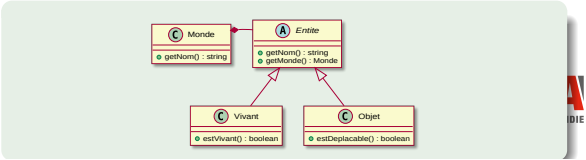
Classe abstraite (CA), pourquoi ?

- Créer une classe dont certaines méthodes sont déclarées mais non définies, mais qui peuvent être utilisées dans des méthodes déclarées et définies
- Créer une classe qui est conceptuelle, qui factorise des attributs ou des méthodes, évitant de les copier/coller dans différentes classes

Conséquences

- Classe qui ne peut pas avoir d'instance (d'où le terme abstrait)
- Classe qui peut avoir des méthodes abstraites : méthodes déclarées mais non définies
- Couplage fort : une classe C est de type CA si elle est sous-classe de CA





Interface (I)

- Pourquoi ?
  - Définir un type avec uniquement des déclarations de méthodes
- Conséquence
  - Il n'est pas possible de définir des méthodes, juste les déclarer
  - Couplage faible : une classe C est de type I si elle implémente I en définissant les méthodes déclarées par I

INSA ROUEN NORMANDIE

Classe abstraite en Python

Comment ?

- Déclarer une classe abstraite :
  - hériter de la classe `ABC` (pour *Abstract Base Class*) proposé par le module `abc`
  - posséder au moins une méthode abstraite (utilisation du décorateur `abstractmethod`). Ce décorateur peut être associé aux décorateurs **classmethod**, **staticmethod** et **property**
- S'assurer que les sous-classes implémentent ces méthodes

Exemple

```python
from abc import ABC, abstractmethod

class Forme(ABC):
    @abstractmethod
    def aire(self) -> float:
        pass

    @abstractmethod
    def perimetre(self) -> float:
        pass
```

```python
class Cercle(Forme):
    def __init__(self, rayon: float):
        self.rayon = rayon

    def aire(self) -> float:
        return 3.14 * self.rayon**2

    def perimetre(self) -> float:
        return 2 * 3.14 * self.rayon
```

Les (`Protocol`)

Constat

- Grâce au *Duck Typing*, le concept d'Interface n'est pas utile en python
- Mais ce n'est qu'à l'exécution que l'on peut se rendre compte des problèmes $\Rightarrow$ besoin de formaliser des dépendances, des attentes

Les protocoles : les interfaces de Python

- Améliorer le typage statique en spécifiant des méthodes requises

Équivalent aux interfaces d'autres langages comme Java

Comment ?

- Déclarer un protocole : sous classe de la classe `Protocol` du module typing
- Être du même type qu'un protocole : juste posséder les méthodes définies par le protocole

Exemple

```python
from typing import Protocol

class PeutVoler(Protocol):
    def voler(self) -> None:
        pass

class Oiseau:
    def voler(self) -> None:
        print("Battre des ailes et voler!")

class Avion:
    def voler(self) -> None:
        print("Planer àtravers les nuages!")

def effectuer_vol(entite: PeutVoler) -> None:
    entite.voler()

oiseau_instance = Oiseau()
avion_instance = Avion()

effectuer_vol(oiseau_instance)
effectuer_vol(avion_instance)
```

INSA ROUEN NORMANDIE

Conclusion

- Les classes `ABC` et `Protocol` offrent des mécanismes puissants pour améliorer la structure et la fiabilité du code en renforçant les contrats
- Les classes abstraites garantissent l'implémentation de méthodes spécifiques dans les sous-classes, renforçant la hiérarchie des classes
- Les protocoles offrent une flexibilité accrue en spécifiant des contrats moins rigides, adaptés à des situations où une hiérarchie stricte n'est pas nécessaire

INSA ROUEN NORMANDIE

Python
Organisation d'un projet python
pip, pyenv, pipenv, git

Sébastien Bonnegent, Nicolas Delestre

Installation de *frameworks*, *packages*, etc.

Pourquoi ne pas utiliser `dpkg`

- Votre programme ne sera pas portable
- La mise à jour de l'OS peut rendre votre programme inutilisable
- L'OS ne propose peut-être pas la dernière version d'une bibliothèque

`pip`

- Python possède son propre gestionnaire de paquets : `pip`
- Il permet d'installer, de mettre à jour de désinstaller des paquets
- Commandes principales :

```
  install      Install packages.
  download     Download packages.
  uninstall    Uninstall packages.
  freeze       Output installed packages in requirements format.
  list         List installed packages.
  show         Show information about installed packages.
...
```

Avantages, inconvénients

Avantages

- Multi plateformes
- Intégré de base à python
- Avoir n'importe quelles versions des bibliothèques

Inconvénients

- Impossible de faire cohabiter des versions différentes de paquets. Cela peut poser des problèmes lorsque des applications python utilisent des paquets avec des versions non compatibles
- Impossible d'utiliser une version spécifique de python

Solution

- Il faudrait pouvoir isoler les applications python du système

Installation

Caractéristiques

- Permet d'utiliser n'importe quelle version de python

Installation

```
$ git clone https://github.com/pyenv/pyenv.git ~/.pyenv
$ echo 'export PYENV_ROOT="$HOME/.pyenv"' >> ~/.bashrc
$ echo 'export PATH="$PYENV_ROOT/bin:$PATH"' >> ~/.bashrc
$ echo -e 'if command -v pyenv 1>/dev/null 2>\&1; then\n \
           eval "$(pyenv init -)"\nfi' >> ~/.bashrc
$ exec "$SHELL"
```

Utilisation

Exemple d'utilisation

```
$ python3 -V
Python 3.6.7
$ pyenv versions
* system (set by /home/bonnegent/.pyenv/version)
$ pyenv install 3.7.3
# ... installation dans ~/.pyenv/versions/3.7.3/

$ pyenv versions
* system (set by /home/bonnegent/.pyenv/version)
  3.7.3
$ pyenv global 3.7.3
$ python3 -V
Python 3.7.3
$ pyenv global system
$ python3 -V
Python 3.6.7
```

venv 1 / 3

Caractéristiques

- Historiquement `virtualenv`, intégré à python à partir de la version 3.4
- Isole les applications python du système d'exploitation
- Utilisation de `pip` pour l'installation des paquets
- Gestion des environnements à l'aide du module `venv`

venv 2 / 3

Création de l'environnement (bonne pratique)

1. Création du répertoire de l'application : `mkdir NomDuProjet && cd NomDuProjet`
2. Création de l'environnement virtuel : `python3 -m venv venv`
   - Création d'un répertoire `venv` dans le projet (avec une arborescence « système » (répertoires bin, include, lib, etc.)
   - Par défaut n'utilise pas les paquets systèmes
3. Activation de l'environnement : `source venv/bin/activate`
   - les variables systèmes sont modifiées
   - le prompt est modifié
4. Mise à jour de pip et installation des paquets
5. Enregistrement de la configuration : `pip freeze > requirements.txt`

venv 3 / 3

Utilisation de l'environnement

- La version de base de `python` est `python3`
- `ipython` n'est pas installé de base (attention son installation avec `pip` donne accès à `ipython` et `ipython3`)
- Configuration si nécessaire de `PYTHONPATH`

Sortir de l'environnement

- Commande : `deactivate`

Reprise d'une configuration

1. Création et activation de l'environnement virtuel (étapes 2 et 3)
2. Installation des paquets :
   `pip install -r requirements.txt`
3. Configuration si nécessaire de `PYTHONPATH`

L'arme ultime

Description

- `https://docs.pipenv.org/en/latest/`
- Surcouche à venv et à pip
- Outil recommandé par python et les mainteneurs de pip
- Gestion automatique du répertoire du virtualenv
- Plusieurs environnements possibles (prod et dev)
- Mise à jour facile
- Contrainte sur la version de python
- Différence entre les dépendances nécessaires et les dépendances des dépendances

Installation

```
$ sudo pip3 install pipenv
$ sudo apt install pipenv
```

Mise en place

Pour un nouveau projet

```
$ pipenv --python 3.7
Virtualenv location: /home/bonnegent/.local/share/virtualenvs/test-hxkKlP5o
Creating a Pipfile for this project...
$ pipenv install ipython
$ pipenv install black==19.3b0 --dev
$ pipenv install pytest --dev
```

Mise à jour

```
$ # affichage des mises à jours disponibles
$ pipenv update --outdated
$ # réalisation des mises à jours (lock + sync)
$ pipenv update
```

Comment ça marche ?

Le fichier Pipfile

```
$ ls
Pipfile  Pipfile.lock
$ cat Pipfile
[[source]]
url = "https://pypi.org/simple"
verify_ssl = true
name = "pypi"

[packages]
ipython = "*"

[dev-packages]
black = "==19.3b0"
pytest = "*"

[requires]
python_version = "3.7"
```

Utilisation

Utilisation

```
$ pipenv shell
$ pipenv run ipython
```

Commandes utiles

```
$ # vérification des mises à jour de sécurité
$ pipenv check
$ # arbre des dépendances
$ pipenv graph
$ # suppression du virtual env
$ pipenv --rm
$ # chemin du virtual env
$ pipenv --venv
$ # installation / mise à jour de l'environnement de production
$ pipenv install
$ # installation / mise à jour de l'environnement de production+dev
$ pipenv install --dev
```

Configuration pour git

Qu'est-ce qui ne doit pas être *pushé* ?

- le virtual env (`.venv`)
- les résultats des compilations (`.pyc`, `__pycache__`, etc)

Comment empêcher automatiquement ces fichiers d'être *pushés*

- en ajoutant un fichier `.gitignore` à la racine du projet
- le dépôt git de git propose des fichiers types pour chaque langage (`https://github.com/github/gitignore`), celui de python prend en compte ces exigences

Organisation d'un projet python

Proposition d'une organisation de fichiers, répertoires

```
NomDuProjet
  + package_racine
    + __init__.py
    + sous_package_1
      + __init__.py
      + ...
    + sous_package_2
      + __init__.py
      + ...
    + module_1
    + ...
  + Pipfile
  + Pipfile.lock
  + tests
    + reprendre la hiérarchie des packages et mettre autant de fichier
test_XXX.py qu'il y a de modules (attention tous avec des noms différents)
  + programme_principal.py (si plusieurs, utilisation d'un répertoire bin)
  + README
  + .gitignore
```

Exemple : le projet CAO 1 / 4

Organisation du projet

```
ASI_CAO
  + cao
    + __init__.py
    + core
      + __init__.py
      + point.py
      + polyligne.py
    + lib
      + __init__.py
      + point.py
      + polyligne.py
  + tests
    + core
      + test_core_point.py
      + test_core_polyligne.py
    + lib
      + test_lib_point.py
      + test_lib_polyligne.py
  + main.py
  + Pipfile
  + Pipfile.lock
```

Exemple : le projet CAO 2 / 4

Création de l'environnement

```
ASI_CAO$ pipenv --python 3.7
```

Installation de pytest

```
ASI_CAO$ pipenv install --dev pytest
Creating a virtualenv for this project...
Pipfile: /tmp/test2/Pipfile
Using /home/bonnegent/.pyenv/versions/3.7.3/bin/python3.7m (3.7.3) to create virtua
Running virtualenv with interpreter /home/bonnegent/.pyenv/versions/3.7.3/bin/pytho
Using base prefix '/home/bonnegent/.pyenv/versions/3.7.3'
/usr/lib/python3/dist-packages/virtualenv.py:1086: DeprecationWarning: the imp modu
  import imp
New python executable in /home/bonnegent/.local/share/virtualenvs/test2-h6PFdzq5/bi
Also creating executable in /home/bonnegent/.local/share/virtualenvs/test2-h6PFdzq5
Installing setuptools, pkg_resources, pip, wheel...done.

Virtualenv location: /home/bonnegent/.local/share/virtualenvs/test2-h6PFdzq5
Creating a Pipfile for this project...
```

Exemple : le projet CAO 3 / 4

C'est python3 qui est le seul python

```
ASI_CAO$ pipenv run python --version
Python 3.7.3
ASI_CAO$
ASI_CAO$ pipenv shell
(venv)ASI_CAO$ python --version
Python 3.7.3
(venv)ASI_CAO$ desactivate
```

Lancement des tests unitaires

```
ASI_CAO$ pipenv run black .
ASI_CAO$ pipenv run pytest .
============================= test session starts ==============================
platform linux -- Python 3.7.3, pytest-3.0.7, py-1.4.33, pluggy-0.4.0
rootdir: .../ASI_CAO, inifile:
collected 18 items

tests/core/test_core_point.py ..
tests/core/test_core_polyligne.py ..........
tests/lib/test_lib_point.py ..
tests/lib/test_lib_polyligne.py ...
```

Exemple : le projet CAO 4 / 4

Utilisation en production

```
# activation dans notre shell
$ export PIPENV_VENV_IN_PROJECT="yes"
# enregistrement
$ echo 'export PIPENV_VENV_IN_PROJECT="yes"' >> /root/.bash_profile
$ pipenv install --ignore-pipfile
$ pipenv --venv
/opt/asi_cao/.venv
```

Python
Tests Unitaires

Nicolas Delestre

INSA
ROUEN NORMANDIE

Plan

INSA ROUEN NORMANDIE

Exemple d'erreur

point.py

```
def _get_x(self):
    return self._x

def _set_x(self, x):
    self._x = x

x = property(_get_x, _set_x)

@property
def y(self):
    return self._x
```

INSA ROUEN NORMANDIE

Plusieurs *frameworks*

- Il y a eu historiquement plusieurs frameworks de développement de tests unitaires sour python :
  - *unittest*
  - nose
  - *doctest*
  - **pytest**

INSA ROUEN NORMANDIE

unittest 1 / 2

Caractéristiques

- S'inspire des framework de tests unitaires des autres langages, comme JUnit
- Intégré de base à python
- Il ne tire pas parti des aspects introspections du python

Développement du test unitaire : test_point.py

```
import unittest
from point import Point2D

class TestPoint2D(unittest.TestCase):
    def test_abscisse(self):
        self.assertEqual(Point2D(1,2).x, 1)

    def test_ordonnee(self):
        self.assertEqual(Point2D(1,2).y, 2)

if __name__ == "__main__":
    unittest.main()
```

unittest 2 / 2

Exécution du test unitaire

```
$ python3 test_point.py
.F
======================================================================
FAIL: test_ordonnee (__main__.TestPoint2D)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_point.py", line 11, in test_ordonnee
    self.assertEqual(Point2D(1,2).y, 2)
AssertionError: 1 != 2

----------------------------------------------------------------------
Ran 2 tests in 0.000s

FAILED (failures=1)
```

INSA ROUEN NORMANDIE

doctest 1 / 5

Caractéristiques

- Très *pythonic*
- Avantages :
  - intégré de base à python
  - la documentation intègre les tests unitaires (tests à jour, sert aussi de documentation)
  - Le module et les tests unitaires forment un tout (exécution des tests dans la partie `if __name__ == "__main__"`)
- Inconvénients :
  - si les tests sont long, la documentation prend beaucoup de place
  - attention aux espaces (ou tabulations) après le résultat attendu

INSA ROUEN NORMANDIE

Définition des tests unitaires : point.py

```
@property
def x(self):
    """ Propriété permettant d'obtenir et de fixer l'abscisse d'un Point2D

    >>> p = Point2D(1,2)
    >>> p.x
    1
    >>> p.x = 3
    >>> p.x
    3
    """
    return self._x

@x.setter
def x(self, x):
    self._x = x
```

Définition des tests unitaires : point.py (suite et fin)

```
    @property
    def y(self):
        """ Propriété permettant d'obtenir et de fixer l'ordonnée d'un Point 2D

        >>> p = Point2D(1,2)
        >>> p.y
        2
        >>> p.y = 3
        >>> p.y
        3
        """
        return self._x
```

Code d'exécution : point.py

```
if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

doctest 4 / 5

Exécution directement depuis le module

```
$ python3 point.py
**********************************************************************
File "point.py", line ?, in __main__.Point2D.y
Failed example:
    p.y
Expected:
    2
Got:
    1
**********************************************************************
File "point.py", line ?, in __main__.Point2D.y
Failed example:
    p.y
Expected:
    3
Got:
    1
**********************************************************************
1 items had failures:
   2 of   4 in __main__.Point2D.y
***Test Failed*** 2 failures.
```

doctest 5 / 5

Apparition des tests dans la documentation

```
$ python3
>>> import point
>>> help(point.Point2D)
Help on class Point2D in module point:

class Point2D(builtins.object)
 |  Methods defined here:
...
 |  Data descriptors defined here:
 |
...
 |  x
 |      Propriété permettant d'obtenir et de fixer l'abscisse d'un Point2D
 |
 |      >>> p = Point2D(1,2)
 |      >>> p.x
 |      1
 |      >>> p.x = 3
 |      >>> p.x
 |      3
 |
 |  y
 |      Propriété permettant d'obtenir et de fixer l'ordonnée d'un Point 2D
 |
 |      >>> p = Point2D(1,2)
 |      >>> p.y
 |      2
 |      >>> p.y = 3
 |      >>> p.y
 |      3
```

Pytest

Caractéristiques

- Le plus populaire !
- Très simple à mettre en oeuvre (utilisation poussée de l'introspection)
- Recherche de lui même les tests unitaires (modules/fonctions dont le nom commence par `test_`)
- Des messages clairs
- Les méthodes avant (`setup()`) et après (`teardown()`) les tests unitaires sont remplacées par des *fixtures*
- Possibilité de paramétrer des tests
- Sait utiliser les tests unittest et doctest
- Ne fait pas parti de l'installation de base :
  `pipenv install pytest`

Premier exemple 1 / 2

test_point.py

```
#!/usr/bin/env python

from point import Point2D

def test_abscisse():
    assert Point2D(1,2).x == 1

def test_ordonnee():
    assert Point2D(1,2).y == 2
```

INSA ROUEN NORMANDIE

Premier exemple 2 / 2

Exécution

```
python3 -m pytest
================================== test session starts ==================================
platform linux -- Python 3.7.2, pytest-5.3.5, py-1.8.1, pluggy-0.13.1
rootdir:/home/delestre/Documents/Cours/ASI/Python/Cours/08-TestsUnitaires/Version1.1/exemples
/pytest/erreursSurPoint
collected 2 items

test_point.py .F                                                                  [100%]

======================================== FAILURES ========================================
_____________________________________ test_ordonnee _____________________________________

    def test_ordonnee():
>       assert Point2D(1,2).y == 2
E       assert 1 == 2
E        +  where 1 = Point2D(1,1).y
E        +    where Point2D(1,1) = Point2D(1, 2)

test_point.py:9: AssertionError
============================== 1 failed, 1 passed in 0.03s ==============================
```

Deuxième exemple

test_polyligne.py (on suppose point.py corrigé)

```
#!/usr/bin/env python3

import pytest
from polyligne import Polyligne, MemePointInterditErreur
from point import Point2D

def test_polyligne_est_fermee():
    polyligne_fermee = Polyligne(True, Point2D(1,1), Point2D(2,2), Point2D(1,2))
    assert polyligne_fermee.est_ferme

def test_polyligne_longueur():
    polyligne_fermee = Polyligne(True, Point2D(1,1), Point2D(2,2), Point2D(1,2))
    assert len(polyligne_fermee) == 3

def test_polyligne_longueur_apres_ajout():
    polyligne_fermee = Polyligne(True, Point2D(1,1), Point2D(2,2), Point2D(1,2))
    longueur = len(polyligne_fermee)
    polyligne_fermee.ajouter(Point2D(0,0))
    assert len(polyligne_fermee) == longueur + 1

def test_polyligne_ajout_avec_erreur():
    polyligne_fermee = Polyligne(True, Point2D(1,1), Point2D(2,2), Point2D(1,2))
    with pytest.raises(MemePointInterditErreur):
        polyligne_fermee.ajouter(Point2D(1,1))
```

Fixture 1 / 3

Remarques sur l'exemple

- Du code a été copié/collé (création de `polyligne_fermee`)
- Une solution serait de faire une variable globale
- Sauf que certains tests la modifieraient (par exemple `est_polyligne_longueur_apres_ajout`)

Fixture

- les *fixtures* permettent de générer des données de tests
- les *fixtures* sont paramétrable à l'aide des paramètres nommés, entre autres :
  - `scope` qui définit la portée de création (session, module, class, function)
  - `params` et le paramètre formel `request` qui possède un champ `param`
- des *fixtures* sont utilisés comme paramètres formels des tests

Fixture 2 / 3

test_polyligne.py

```
@pytest.fixture(scope="function")
def polyligne_fermee():
    return  Polyligne(True, Point2D(1,1), Point2D(2,2), Point2D(1,2))

@pytest.fixture(scope="function", params=[Point2D(1,1), Point2D(2,2), Point2D(1,2)])
def point_a_ajouter_qui_pose_probleme(request):
    return request.param

@pytest.fixture(scope="function", params=[Point2D(0,0), Point2D(2,1), Point2D(3,3)])
def point_a_ajouter_qui_ne_pose_pas_probleme(request):
    return request.param

def test_polyligne_est_fermee(polyligne_fermee):
    assert polyligne_fermee.est_ferme

def test_polyligne_longueur_apres_ajout(polyligne_fermee):
    longueur = len(polyligne_fermee)
    polyligne_fermee.ajouter(Point2D(0,0))
    assert len(polyligne_fermee) == longueur + 1

def test_polyligne_ajout_avec_erreur(polyligne_fermee, point_a_ajouter_qui_pose_probleme):
    with pytest.raises(MemePointInterditErreur):
        polyligne_fermee.ajouter(point_a_ajouter_qui_pose_probleme)

def test_polyligne_ajout_sans_erreur(polyligne_fermee, point_a_ajouter_qui_ne_pose_pas_probleme):
    polyligne_fermee.ajouter(point_a_ajouter_qui_ne_pose_pas_probleme)
```

Fixture 3 / 3

Exécution : il y a bien 8 tests unitaires qui sont exécutés

```
$ python3 -m pytest
================================== test session starts ==================================
platform linux -- Python 3.7.2, pytest-5.3.5, py-1.8.1, pluggy-0.13.1
rootdir: /home/delestre/Documents/Cours/ASI/Python/Cours/08-TestsUnitaires/Version1.1/exemples
/pytest/erreursSurPolyligne
collected 11 items

test_polyligne.py ...........                                                    [100%]

================================== 11 passed in 0.03s ===================================
```

Paramètre 1 / 3

Tests unitaires paramétrés

- Quelques fois on veut tester une fonction ou une méthode avec plusieurs valeurs
- Il est possible de définir une fonction de tests qui possède un ou plusieurs paramètres formels et de demander l'exécution de cette fonction, grâce au décorateur `@pytest.mark.parametrize`, avec une liste de paramètres effectifs (qui sont des tuples si la fonction admet plusieurs paramètres formels)

Exemple

- On voudrait vérifier que la longueur d'une polyligne fonctionne bien après la création de la dîte polyligne :
  - quelles soient fermées ou pas
  - qu'il y ait utilisation ou pas des paramètres non nommés optionnels

```
def __init__(self, est_ferme, pt1, pt2, *args):
```

Paramètre 2 / 3

test_polyligne.py : ajout d'un test paramétré

```
@pytest.mark.parametrize("polyligne, longueur",
                         [(Polyligne(False, Point2D(1,1), Point2D(2,2)), 2),
                          (Polyligne(True, Point2D(1,1), Point2D(2,2)), 2),
                          (Polyligne(False, Point2D(1,1), Point2D(2,2), Point2D(1,2)), 3)
                        ])
def test_polyligne_longueur_apres_init(polyligne, longueur):
    assert len(polyligne) == longueur
```

INSA ROUEN NORMANDIE

Paramètre 3 / 3

Exécution : il y a bien 11 (8+3) tests unitaires qui sont exécutés

```
$ python -m pytest
================================== test session starts ==================================
platform linux -- Python 3.7.2, pytest-5.3.5, py-1.8.1, pluggy-0.13.1
rootdir: /home/delestre/Documents/Cours/ASI/Python/Cours/08-TestsUnitaires/Version1.1/exemples
/pytest/erreursSurPolyligne
collected 11 items

test_polyligne.py ...........                                                    [100%]

================================== 11 passed in 0.02s ===================================..
```

Python
Les flux

Nicolas Delestre

INSA
ROUEN NORMANDIE

Plan

INSA ROUEN NORMANDIE

Séquences d'octets

Qu'est-ce ?

- Jusqu'à présent les séquences étaient des séquences d'objets
- Python propose deux types séquences d'octets :
  - `bytes` : des séquences immuables d'octets
  - `bytearray` : des séquences d'octets
- Les valeurs des éléments de ces séquences sont donc des naturels compris entre 0 et 255 (levée d'un exception `ValueError` sinon)

INSA ROUEN NORMANDIE

bytes 1 / 3

Constantes

- Les constantes `bytes` sont représentées comme des chaînes de caractères avec la lettre `b` comme préfixe telles qu'elles contiennent :
  - des caractères ASCII pour des valeurs comprises entre 32 et 127
  - le caractère d'échappement \ suivi de la valeur (en décimal, octal ou hexadécimal) pour les autres

```
In [1]: a=b"abc"
In [2]: a
Out[2]: b'abc'
In [3]: a[0]
Out[3]: 97
In [4]: a=b"abcé"
  File "<ipython-input-4-3490d5b33a37>", line 1
    a=b"abcé"
      ^
SyntaxError: bytes can only contain ASCII literal characters.
In [5]: a=b"abc\xF10"

In [6]: a
Out[6]: b'abc\xf10'
```

bytes 2 / 3

Méthodes de classe `fromhex` et d'instance `hex`

- Elles permettent de créer un `bytes` à partir d'une chaîne de caractères contenant des représentations hexadécimales des octets (méthode `fromhex`) ou d'obtenir cette chaîne à partir d'un `bytes` (méthode `hex`)
- Pour `fromhex` les espaces ne sont pas pris en compte et une exception `ValueError` peut être levée

```
In [1]: bytes.fromhex('2Ef0 F1f2 ')
Out[1]: b'.\xf0\xf1\xf2'

In [2]: b'.\xf0\xf1\xf2'.hex()
Out[2]: '2ef0f1f2'
```

INSA ROUEN NORMANDIE

bytes 3 / 3

Constructeur

- Le constructeur de `bytes` peut être :
  - un entier positif n, on obtient alors une séquence de longueur n contenant que des 0
  - un itérable produisant des valeurs comprises entre 0 et 255, on obtient alors un `bytes` de même longueur avec les octets correspondants

```
In [1]: bytes(10)
Out[1]: b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'

In [2]: bytes(range(10))
Out[2]: b'\x00\x01\x02\x03\x04\x05\x06\x07\x08\t'
```

INSA ROUEN NORMANDIE

Classe bytearray

- Les instances de la classe bytearray s'utilisent comme ceux la classe bytes mais elles sont en plus mutables

```
In [1]: a=bytearray(10)

In [2]: a
Out[2]: bytearray(b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00')

In [3]: a.append(1)

In [4]: a
Out[4]: bytearray(b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x01')

In [5]: a[0]=1

In [6]: a
Out[6]: bytearray(b'\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\x01')
```

Méthodes communes

https://docs.python.org/3/library/stdtypes.html

- Ces deux classes proposent un grand nombre de méthodes permettant :
  - de compter le nombre d'occurrences d'une suite d'octets
  - de retrouver un suite d'octets
  - d'ajouter, de remplacer, de découper une suite d'octets
  - de modifier une suite d'octets
  - de questionner une suite d'octets

INSA ROUEN NORMANDIE

Le module io 1 / 6

Le module io

- Le module io permet de gérer trois catégories de flux d'entrée/sortie :

  Text I/O des flux qui attendent et qui produisent des chaîne de caractères (str)

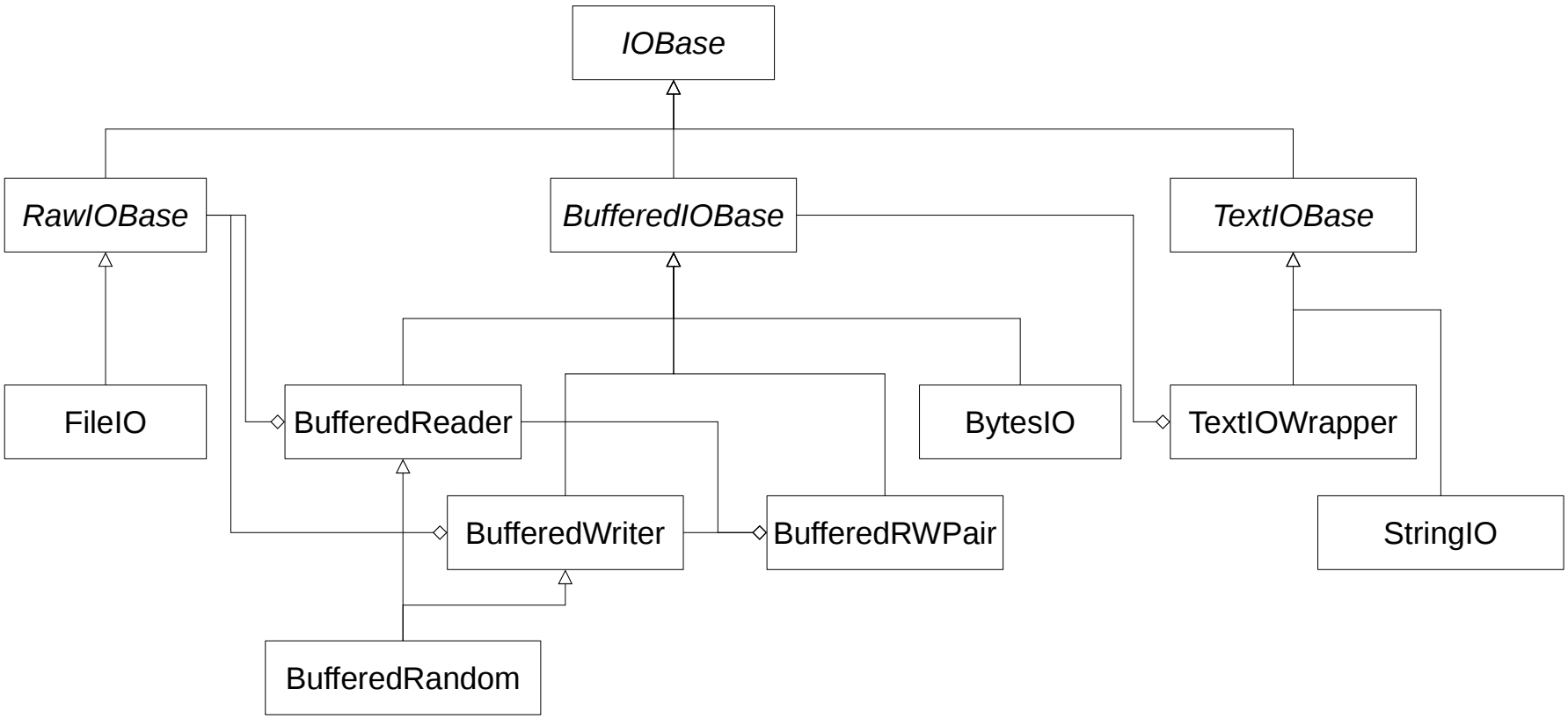
  Binary I/O des flux qui attendent et qui produisent des séquences d'octets (bytes et bytearray)

  Raw I/O des flux bas niveau, orientés bloc, encapsulés dans les flux textes ou binaires

INSA ROUEN NORMANDIE

Le module io 2 / 6

Hiérarchie de classes du module io (version simplifiée)



Le module io 3 / 6

Classe IOBase

- « Classe abstraite, pas de constructeur public »
- Elle définit différentes méthodes qui sont spécialisées par les sous classes
- Elle ne déclare pas les méthodes `read()`, `readinto()`, or `write()` car leurs signatures varient trop
- Les exceptions `ValueError` et `UnsupportedOperation` peuvent être levées

Classe RawIOBase

- « Classe abstraite, pas de constructeur public »
- Elle définit les méthodes de base de lecture et d'écriture d'octets

INSA ROUEN NORMANDIE

Le module io 4 / 6

Classe FileIO

- Elle permet de manipuler des fichiers systèmes
- L'exception FileExistsError peut être levée

Classe BufferedIOBase

- « Classe abstraite, pas de constructeur public »
- Elle met en place un système de cache permettant d'optimiser les lectures ou écritures
- L'exception BlockingIOError peut être levée

Classe BytesIO

- Elle permet de gérer un flux d'octets en mémoire
- La zone mémoire utilisée est supprimée à l'appel de la méthode close

Le module io 5 / 6

Classes BufferedReader, BufferedWriter et BufferedRandom

- Elles permettent d'utiliser des flux bufferisés en lecture, écriture ou lecture/écriture

Classe BufferedRWPair

- Elle permet d'utiliser deux flux bufferisés, l'un en lecture, l'autre en écriture

Classe TextIOBase

- « Classe abstraite, pas de constructeur public »
- Elle gère la persistance de chaînes de caractères encodées

Classe TextIOWrapper

- Elle gère la persistance de chaînes de caractères encodées dans des flux bufferisés

Le module io 6 / 6

Classe StringIO

- Elle gère la persistance de chaînes de caractères encodées en mémoire

Quelques méthodes et propriétés

https://docs.python.org/3/library/io.html

IOBase

close, closed, __enter__, __exit__, fileno, flush, isatty, __iter__, __next__, readable, readline, readlines, seek, seekable, truncate, tell, writable, writelines

RawIOBase

read, readall, readinto, write

BufferedIOBase

detach, read, readinto, read1, write

TextIOBase

detach, encoding, errors, newlines, read, readline, write

Le module sys

- Le module sys propose trois flux :
  - `stdin` qui est un `TextIOWrapper` ouvert en lecture avec un encodage dépendant du système (aujourd'hui très certainement UTF-8)
    - `input` utilise ce flux
  - `stdout` qui est un `TextIOWrapper` ouvert en écriture avec un encodage dépendant du système (aujourd'hui très certainement UTF-8)
    - `print` et `input` utilisent ce flux
  - `stderr` idem `stdout`

INSA ROUEN NORMANDIE

Les fichiers 1 / 3

Ouverture d'un fichier

- Réalisée à l'aide de la fonction `open`
  - Utilise le motif de conception *Factory* : la classe de l'objet retournée varie en fonction des paramètres (du type -texte ou binaire- du mode -lecture, écriture, ajout, etc.-), par exemple :
    - `TextIOWrapper` pour un fichier texte
    - `BufferedReader` pour un fichier binaire ouvert en lecture
    - `BufferedWriter` pour un fichier binaire ouvert en écriture
    - `BufferedRandom` pour un fichier binaire ouvert en lecture/écriture
- L'instance retournée est un *context manager*, donc utilisable avec l'instruction `with...as`

INSA ROUEN NORMANDIE

Les fichiers 2 / 3

Paramètres (version simplifiée)

```
open(file, mode='r', buffering=-1, encoding=None, errors=None,
newline=None)
```

file	chemin du fichier (`str`, `bytes`, `os.PathLike`)
mode	chaîne de caractères composée de : `'r'` en lecture (par défaut), `'w'` en écriture, `'x'` création exclusive (exception si fichier existe déjà), `'a'` en ajout, `'b'` en binaire, `'t'` en mode texte (par défaut), `'+'` en mise à jour (lecture écriture)
buffering	pas de buffer (0 en mode binaire, 1 en mode texte), sinon taille du buffer (la valeur par défaut essaye de faire au mieux)
encoding	en mode texte uniquement
errors	en mode texte pour indiquer comment gérer les erreurs d'encodage (`'strict'`, `'ignore'`, `'replace'`, etc.)
newline	en mode texte uniquement

INSA ROUEN NORMANDIE

Les fichiers 3 / 3

points.py

```
    @classmethod
    def enregistrer_les_points(cls, flux: io.TextIOBase):
        for pt in cls._les_points:
            flux.write("%d %d\n" % (pt.x, pt.y))
```

```
In [3]: from point import Point2D

In [4]: pt1=Point2D(1,1)

In [5]: pt2=Point2D(2,2)

In [6]: with open("/tmp/points.txt", "w") as f:
    Point2D.enregistrer_les_points(f)

In [7]:
Do you really want to exit ([y]/n)? y
$ cat /tmp/points.txt
1 1
2 2
```

Les modules `pickle` et `json`

La sérialisation d'objets

- Le module `pickle` propose des fonctions permettant de sérialiser (respectivement désérialiser) des objets dans un flux binaire qui doit proposer la méthode `write` (respectivement `read`)
- Le module `json` propose des fonctions permettant de sérialiser et désérialiser des objets dans un flux texte
- Méthodes `dump` et `dumps` pour la sérialisation
- Méthodes `load` et `loads` pour la désérialisation

INSA ROUEN NORMANDIE

Python
Les logs en python

Nicolas Delestre

INSA
ROUEN NORMANDIE

Les logs

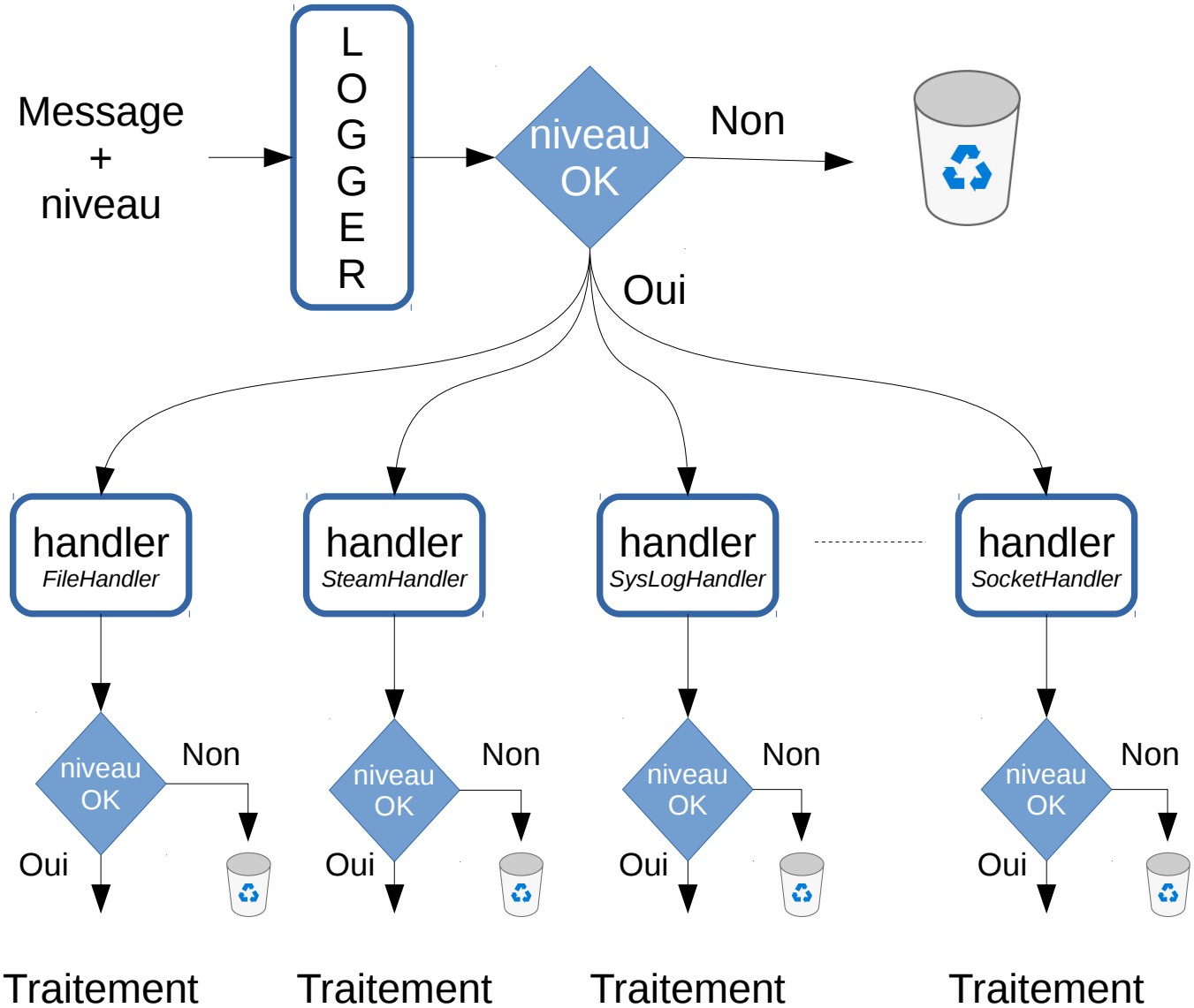
Lorsque l'on veut débugger un programme...

- On utilise souvent `print`, mais ce n'est pas satisfaisant car :
  - les informations sont envoyées sur la sortie standard
  - on mélange information sur le fonctionnement du programme et utilisation normal de la sortie standard
  - il n'y a pas forcément de sortie standard (programme python utilisé comme démon)
  - les messages ne sont pas formatés
  - on ne sait pas distinguer ce qui est important de ce qui ne l'est pas

logging

- `logging` est un framework standard de python
- il s'inspire de Log4j du monde Java (identifiants en *CamelCase* :-( )
- il permet de *formater* les messages
- il permet de conditionner les messages
- il permet d'envoyer les messages sur un ou plusieurs flux (*handler*)

Fonctionnement de `logging`



inspiré de *http://sametmax.com/ecrire-des-logs-en-python/*

Les niveaux

- Il y a 5 niveaux : `DEBUG` < `INFO` < `WARNING` < `ERROR` < `CRITICAL`
- Un niveau est associé :
  - à un message
  - au *logger*
  - aux *handler*
- Le message est généré dans le flux associé à un *handler* si son niveau est supérieur ou égal au niveau du *logger* **et** du *handler*

INSA ROUEN NORMANDIE

Utilisation d'un *logger* dans votre code

Étapes

1. Importer le module `logging`
2. Obtenir le *logger* : fonction `getLogger`
3. Utiliser la méthode :
   - `log` avec comme premier paramètre effectif le niveau de log et comme deuxième le message
   - `debug`, `info`, `warning`, `error` ou `critical` avec comme paramètre effectif le message

INSA ROUEN NORMANDIE

Paramétrage du logger et création des *handler*

- À réaliser dans votre programme principal en fonction :
  - de paramètres/options données à l'exécution du programme
  - d'un fichier de configuration

Étapes

1. Importer le module `logging`
2. Obtenir le *logger* : fonction `getLogger`
3. Fixer le niveau de logger : méthode `setLevel` avec les constantes du module `logging`
4. Créer un *handler*, plusieurs classes disponibles : `StreamHandler` (sortie erreur standard), `FileHandler` (fichier de log), etc.
5. Paramétrer le *handler* : `setLevel` (niveau), `setFormatter` (formatage), etc.
6. Ajouter le *handler* au *logger* : méthode `addHandler`

Les étapes 4 à 6 peuvent être répétées plusieurs fois

Exemple 1 / 11

Des logs pour notre application sur les polylignes

On va utiliser trois niveaux de log :

- `debug` pour les créations et modification
- `info` pour les getter
- `error` lors des levées d'exception

`polyligne.py`

```
4 import logging
5
6 logger = logging.getLogger()
```

INSA ROUEN NORMANDIE

Exemple 2 / 11

Extrait de polyligne.py

```
12      @staticmethod
13      def _verifier_pt_ou_pts_absent(pt_ou_pts, pts):
14          try:
15              iter(pt_ou_pts)
16          except Exception:
17              if pt_ou_pts in pts:
18                  logger.error("Exception MemePointInterditErreur %s appartient déjà  à
    %s" % (pt_ou_pts, self))
19                  raise MemePointInterditErreur(repr(pt_ou_pts) + " appartient déja à
    la polyligne")
20              return
21          for pt in pt_ou_pts:
22              if pt in pts:
23                  logger.error("Exception MemePointInterditErreur %s appartient déjà  à
    %s" % (pt_ou_pts, self))
24                  raise MemePointInterditErreur(repr(pt) + " appartient déja à la polyligne")
```

INSA ROUEN NORMANDIE

Exemple 3 / 11

Extrait de polyligne.py

```
12    def __init__(self, est_ferme, pt1, pt2, *args):
13        logger.debug("Creation d'une polyligne : est_ferme : %s, points : %s" % (est_ferme,
14                                                                                  [pt1, pt2]))
15        if pt1 == pt2:
16            raise MemePointInterditErreur("pt1 et pt2 doivent être différents")
17        self._est_ferme = est_ferme
18        self._points = [pt1, pt2]
19        for pt in args:
20            self.ajouter(pt)
```

```
36    def _get_est_ferme(self):
37        logger.info("appel à est_ferme")
38        return self._est_ferme
39
40    def _set_est_ferme(self, est_ferme):
41        logger.debug("%s de la polyligne %s" % ("Fermeture" if est_ferme else "Ouverture",
    self))
42        self._est_ferme = est_ferme
43
44    est_ferme = property(_get_est_ferme, _set_est_ferme)
```

Exemple 4 / 11

Extrait de polyligne.py

```
77      def __iter__(self):
78          logger.info("Début d'une itération")
79          for pt in self._points:
80              logger.info("obtention de %s dans une itération" % pt)
81              yield pt
```

Un script qui utilise Polyligne : exemple_sans_logging.py

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

if __name__ == "__main__":
    from polyligne import Polyligne
    from point import Point2D

    pl = Polyligne(True, Point2D(0,0), Point2D(1,1), Point2D(2,0))
    for pt in pl:
        print(pt)
```

Résultats

```
$ python3 exemple_sans_logging.py
(0, 0)
(1, 1)
(2, 0)
```

INSA ROUEN NORMANDIE

Exemple 6 / 11

logging_sortie_erreur_standard.py

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

import logging

logger = logging.getLogger()

sortie_standard = logging.StreamHandler()
sortie_standard.setLevel(logging.INFO)
logger.addHandler(sortie_standard)
```

INSA ROUEN NORMANDIE

exemple_logging_un_seul_handler.py

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

if __name__ == "__main__":
    from polyligne import Polyligne
    from point import Point2D

    import logging
    import logging_sortie_erreur_standard

    logger = logging.getLogger()
    logger.setLevel(logging.DEBUG)

    pl = Polyligne(True, Point2D(0,0), Point2D(1,1), Point2D(2,0))
    for pt in pl:
        print(pt)
```

Exemple 8 / 11

Résultats

```
$ python3 exemple_logging_un_seul_handler.py
Début d'une itération
obtention de (0, 0) dans une itération
obtention de (1, 1) dans une itération
Début d'une itération
obtention de (0, 0) dans une itération
(0, 0)
obtention de (1, 1) dans une itération
(1, 1)
obtention de (2, 0) dans une itération
(2, 0)
```

Résultats en redirigeant le flux d'erreur standard

```
$ python3 exemple_logging_un_seul_handler.py 2> sortie_erreur_standard.txt
(0, 0)
(1, 1)
(2, 0)
```

ROUEN NORMANDIE

logging_fichier_temp.py

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

import logging
import __main__ as main

logger = logging.getLogger()

formatter = logging.Formatter('%(asctime)s :: %(levelname)s :: %(message)s')

fichier_temp = logging.FileHandler("/tmp/" + main.__file__[:-3] + ".log")
fichier_temp.setLevel(logging.DEBUG)
fichier_temp.setFormatter(formatter)
logger.addHandler(fichier_temp)
```

INSA ROUEN NORMANDIE

exemple_logging_deux_handlers.py

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

if __name__ == "__main__":
    from polyligne import Polyligne
    from point import Point2D

    import logging
    import logging_sortie_erreur_standard
    import logging_fichier_temp

    logger = logging.getLogger()
    logger.setLevel(logging.DEBUG)

    pl = Polyligne(True, Point2D(0,0), Point2D(1,1), Point2D(2,0))
    for pt in pl:
        print(pt)
```

Exemple 11 / 11

Résultats

```
$ python3 exemple_logging_deux_handlers.py
Début d'une itération
obtention de (0, 0) dans une itération
obtention de (1, 1) dans une itération
Début d'une itération
obtention de (0, 0) dans une itération
(0, 0)
obtention de (1, 1) dans une itération
(1, 1)
obtention de (2, 0) dans une itération
(2, 0)
$ cat /tmp/exemple_logging_deux_handlers.log
2018-04-04 08:39:12,461 :: DEBUG :: Creation d'une polyligne : est_ferme : True, points : [Point2D(0,0), Point
2018-04-04 08:39:12,462 :: INFO :: Début d'une itération
2018-04-04 08:39:12,462 :: INFO :: obtention de (0, 0) dans une itération
2018-04-04 08:39:12,462 :: INFO :: obtention de (1, 1) dans une itération
2018-04-04 08:39:12,462 :: DEBUG :: Ajout des points [Point2D(2,0)] à la polyligne ((0, 0), (1, 1))
2018-04-04 08:39:12,463 :: INFO :: Début d'une itération
2018-04-04 08:39:12,463 :: INFO :: obtention de (0, 0) dans une itération
2018-04-04 08:39:12,463 :: INFO :: obtention de (1, 1) dans une itération
2018-04-04 08:39:12,463 :: INFO :: obtention de (2, 0) dans une itération
```

INSA ROUEN NORMANDIE

Python
Debuggeur

Nicolas Delestre

INSA
ROUEN NORMANDIE

Debugguer

Pourquoi utiliser un debuggeur ?

- Les logs permettent de contrôler l'exécution général d'un programme
- Ils ne permettent pas de débugger un algorithme
- Les `print` sont à proscrire

pdb3 (ou ipdb3)

- C'est à la fois un outil et un module
- Il est interactif (ipdb l'est encore plus)
- Il est fourni de base avec python

INSA ROUEN NORMANDIE

Code pour la démonstration 1 / 3

lib_polyligne.py

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

from polyligne import Polyligne
from point import Point2D
import lib_point as lp

def est_a_l_interieur(pt: Point2D, pl: Polyligne) -> bool:
    def angle(pt, pt1, pt2):
        vect1 = lp.vecteur(pt, pt1)
        vect2 = lp.vecteur(pt, pt2)
        if vect1 != vect2:
            return lp.angle(vect1, vect2)
        else:
            return 0

    somme_angles = 0
    pt_courant = pl[0]
    for pt_suivant in pl[1:]:
        somme_angles += angle(pt, pt_courant, pt_suivant)
        pt_courant = pt_suivant
    somme_angles += angle(pt, pt_courant, pl[0])
    return abs(abs(somme_angles) - 3.14159) < 1e-3 # C'est ici le bug 2 * 3.14159
```

Code pour la démonstration 2 / 3

exemple_utilisation_polyligne.py

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

from polyligne import Polyligne
from point import Point2D
from lib_polyligne import est_a_l_interieur

pl = Polyligne(True, Point2D(0,0), Point2D(1,1), Point2D(2,0))
for pt in pl:
    print(pt)
print(est_a_l_interieur(Point2D(1, 0.5), pl))
```

Exécution du script

```
$ python3 exemple_utilisation_polyligne.py
(0, 0)
(1, 1)
(2, 0)
False
```

Code pour la démonstration 3 / 3

Lancement du debuggeur

```
$ ipdb3 exemple_utilisation_polyligne.py
> ... /exemple_utilisation_polyligne.py(4)<module>()
      3
----> 4 from polyligne import Polyligne
      5 from point import Point2D

ipdb>
```

INSA ROUEN NORMANDIE

Les commandes 1 / 4

Affichage

h[com]	*help*, affiche toute les commandes (si suivi d'une commande affiche l'aide de la commande)
l	*list*, affiche quelques lignes de code avant et après l'instruction courante
ll	*long list*, affiche le code du module (ou de la fonction) courant
w	*where*, affiche la pile des appels
p	*print*, suivi de `locals()` (ou `globals()`) permet d'afficher toutes les variables locales (ou globales).
pp	*pretty print*, *idem*
a	*args*, affiche les arguments de la fonction courante
display [expr]	affiche une expression dès qu'elle change (si pas d'argument, affiche toutes les expressions)
undisplay [expr]	arrêt l'affichage d'une ou de toutes les expressions

Les commandes 2 / 4

Exécution

n	*next*, exécute l'instruction courante
s	*step in*, « rentre » dans l'instruction courante
j num	*jump*, va directement à la ligne `num`
unt num	*until*, exécute les instructions jusqu'à ce que leurs numéros de ligne soit $\geq$ `num`
r	*return*, exécute toutes les instructions de la fonction courante
c	*continue*, continue l'exécution du programme normalement jusqu'au prochain point d'arrêt
run|restart [args ...]	relance le debuggeur

INSA ROUEN NORMANDIE

Points d'arrêts

b[[fic :](num|func) [, cond]] *breakpoint*, met un point d'arrêt au fichier *fic* à la ligne numéro *num* ou fonction *fun*. Ce point d'arrêt peut être conditionné par *cond*. Si aucun numéro de ligne ou nom de fonction, alors affiche la liste des points d'arrêt

tbreak *temporary break*, *idem break* mais disparaît au premier arrêt

condition bp [cond] fixe ou annule une condition d'un point d'arrêt

disable [bp [bp ..]] désactive un ou plusieurs points d'arrêt

enable [bp [bp ..]] active un ou plusieurs points d'arrêt

cl [fic :num|bp [bp ...]] *clear*, supprime tous ou quelques points d'arrêt

commands bp [cmd1 ; cmd2... end] associe une liste de commandes (souvent de l'affichage) à un point d'arret

INSA ROUEN NORMANDIE

Alias

alias [nom [instr]] permet d'ajouter (ou de remplacer une commande du debuggeur) en y associant une instruction python. Cette commande peut être paramétrée (utilisation de %1, %2, etc. %*).

unalias nom supprime un alias

Modification de l'état

- Il est possible de modifier l'état du programme par des affectations
- Attention les commandes sont prioritaires (si utilisation d'une variable p dans le code, on ne peut pas faire `p = 0`). Utilisation dans ce cas du préfixe `!` pour indiquer au debuggeur d'utiliser explicitement la variable `!p = 0`.

INSA ROUEN NORMANDIE

PDB et les tests unitaires 1 / 3

test_lib_polyligne.py

```python
#!/usr/bin/python3
# -*- coding: utf-8 -*-

import pytest
from polyligne import Polyligne
from point import Point2D
from lib_polyligne import est_a_l_interieur

@pytest.mark.parametrize("polyligne, pt, resultat_attendu",
                         [(Polyligne(True, Point2D(0, 0), Point2D(1, 1),
                                     Point2D(2, 0)), Point2D(1, 0.5), True),
                          (Polyligne(True, Point2D(0, 0), Point2D(1, 1),
                                     Point2D(2, 0)), Point2D(0.5, 1), False),
                          (Polyligne(True, Point2D(0, 0), Point2D(1, 1),
                                     Point2D(2, 0)), Point2D(0.5, 0.5), False),
                        ])
def test_est_a_l_interrieur(polyligne, pt, resultat_attendu):
    assert est_a_l_interieur(pt, polyligne) == resultat_attendu
```

INSA ROUEN NORMANDIE

PDB et les tests unitaires 2 / 3

Exécution du test unitaire

```
$ py.test-3
===== test session starts =======
platform linux -- Python 3.5.2, pytest-2.8.7, py-1.4.31, pluggy-0.3.1
rootdir: .../Cours/10-PDB/Version1.0/exemples, inifile:
collected 3 items

test_lib_polyligne.py F..

==== FAILURES ==================
___ test_est_a_l_interrieur[polyligne0-pt0-True] ________

polyligne = Polyligne(True, Point2D(0.000000,0.000000), Point2D(1.000000,1.000000), Point2D(2.000000,0.000000)
pt = Point2D(1.000000,0.500000), resultat_attendu = True

    @pytest.mark.parametrize("polyligne, pt, resultat_attendu",
                             [(Polyligne(True, Point2D(0, 0), Point2D(1, 1),
                                         Point2D(2, 0)), Point2D(1, 0.5), True),
                              (Polyligne(True, Point2D(0, 0), Point2D(1, 1),
                                         Point2D(2, 0)), Point2D(0.5, 1), False),
                              (Polyligne(True, Point2D(0, 0), Point2D(1, 1),
                                         Point2D(2, 0)), Point2D(0.5, 0.5), False),
                             ])
    def test_est_a_l_interrieur(polyligne, pt, resultat_attendu):
>       assert est_a_l_interieur(pt, polyligne) == resultat_attendu
E       assert est_a_l_interieur(Point2D(1.000000,0.500000), Polyligne(True, Point2D(0.000000,0.000000), Point

test_lib_polyligne.py:18: AssertionError
```

Lancement du debuggeur au premier test unitaire échoué

- option `--pdb` à `py.test`
- utilisation de la commande *debug* qui permet de débugger une expression python, celle qui a fait échouer le test (utilisation du debugger dans le debugger).

INSA ROUEN NORMANDIE

Le module pdb

En fait. . .

- pdb est un module
- il propose entre autres les fonctions (paramètres formels non précisés ici, voir la documentation [a]) :

`run`	pour débugger une expression python représentée par une chaîne
`runcall`	pour debugger une fonction
`set_trace`	pour insérer dans le code un point d'arrêt

a. `https://docs.python.org/3.6/library/pdb.html`

INSA ROUEN NORMANDIE

Python
Arcanes du langage

Nicolas Delestre

INSA
ROUEN NORMANDIE

Cycle de vie d'un objet

La création

- Rappel : `__init__` ne crée pas l'objet, elle l'initialise, c'est `__new__` qui le créé
- « On utilise rarement `__new__`. Les deux cas principaux sont :
  - si on hérite d'un type immutable (`str`, `int`, `tuple`, etc), `__new__` est le seul endroit où on puisse initialiser l'objet.
  - dans le cas des métaclasses. »[a]

a. `http://sametmax.com/la-difference-entre-__new__-et-__init__-en-python/`

La suppression d'un objet

- Lorsque le *garbage collector* supprime un objet, la méthode `__del__` est appelée juste avant

INSA ROUEN NORMANDIE

La gestion des attributs 1 / 4

- `getattr(obj, nom)`, `setattr(obj, nom, valeur)` et `delattr(obj, nom, valeur)` permettent d'utiliser l'attribut `nom` d'`obj` (objet, classe, module), par exemple :
  - la demande d'interprétation de `o.a` exécute `getattr(o,a)`
- `__getattribute__(self, nom)`, `__getattr__(self, nom)`, `__setattr__(self, nom, valeur)` et `__delattr__(self, nom)` permettent de modifier le comportement de l'utilisation des attributs
  - l'exécution `getattr(o,a)` exécute `o.__getattribute__(a)`. Si `a` n'est pas déclaré pour `o`, alors c'est `o.__getattr__(a)` qui est exécuté (par défaut levé de l'exception `AttributeError`)
- Les attributs d'un objet sont stockés dans l'attribut `__dict__`
- Un attribut `__a` d'une classe `C` est nommé `_C__a`

INSA ROUEN NORMANDIE

La gestion des attributs 2 / 4

Script attributs.py

```
1  #!/usr/bin/env python
2
3  class Foo(object):
4      def __init__(self):
5          self.a = 1
6          self.__b = 2
7
8      def __getattribute__(*args):
9          print(f"Appel de __getattribute__: {args}")
10         return object.__getattribute__(*args)
11
12     def __setattr__(*args):
13         print(f"Appel de __setattr__: {args}")
14         return object.__setattr__(*args)
15
16     def m(self):
17         print("Appel de m")
18         print(f"a : {self.a}")
19         print(f"__b : {self.__b}")
20
21 if __name__ == "__main__":
22     o = Foo()
23     o.m()
24     print(f"o.__dict__ : {o.__dict__}")
25     print(f"Foo.__dict__ : {Foo.__dict__}")
26     print(f"o.a : {o.a}")
27     print(f"o.__b : {o.__b}")
```

La gestion des attributs 3 / 4

Exécution du script attributs.py

```
$ python attributs.py
Appel de __setattr__: (<__main__.Foo object at 0x7f866de2f940>, 'a', 1)
Appel de __setattr__: (<__main__.Foo object at 0x7f866de2f940>, '_Foo__b', 2)
Appel de __getattribute__: (<__main__.Foo object at 0x7f866de2f940>, 'm')
Appel de m
Appel de __getattribute__: (<__main__.Foo object at 0x7f866de2f940>, 'a')
a : 1
Appel de __getattribute__: (<__main__.Foo object at 0x7f866de2f940>, '_Foo__b')
__b : 2
Appel de __getattribute__: (<__main__.Foo object at 0x7f866de2f940>, '__dict__')
o.__dict__ : {'a': 1, '_Foo__b': 2}
Foo.__dict__ : {'__module__': '__main__',
  '__init__': <function Foo.__init__ at 0x7f0be9ead950>,
  '__getattribute__': <function Foo.__getattribute__ at 0x7f0be9ead9d8>,
  '__setattr__': <function Foo.__setattr__ at 0x7f0be9eada60>,
  'm': <function Foo.m at 0x7f0be9eadae8>,
  '__dict__': <attribute '__dict__' of 'Foo' objects>,
  '__weakref__': <attribute '__weakref__' of 'Foo' objects>,
  '__doc__': None}
```

Exécution du script attributs.py (suite)

```
Appel de __getattribute__: (<__main__.Foo object at 0x7f866de2f940>, 'a')
o.a : 1
Appel de __getattribute__: (<__main__.Foo object at 0x7f866de2f940>, '__b')
Traceback (most recent call last):
  File "prive.py", line 26, in <module>
    print(f"o.__b : {o.__b}")
  File "prive.py", line 10, in __getattribute__
    return object.__getattribute__(*args)
AttributeError: 'Foo' object has no attribute '__b'
```

INSA ROUEN NORMANDIE

Métaclasse

- Tout est objet en python. Les classes sont des objets. Les classes sont donc instances de classe que l'on nomme métaclasse. On peut créer des métaclasses pour ajouter ou modifier le comportement de classes.
- `type` est la métaclasse par défaut :
  - `type` est sa propre instance
  - `type` est *appelable* : la création d'une classe appelle en fait la fonction `type(nom, super_classes, dictionnaire_attributs)`
  - On crée une nouvelle métaclasse en créant une sous classe de `type`
- On créé une classe `C` de métaclasse `M` grâce au paramètre formel `metaclass` :

```
class C(object, metaclass=M)
```

Méta-programmation 2 / 7

Pourquoi créer une métaclasse ?

- On crée des métaclasses lorsque l'on veut modifier le comportement des classes :
  - durant leur création : `__new__`, `__init__`
  - durant la création de leurs objets : `__call__`
  - durant l'accès à leurs attributs de classe : `__getattr__`, `__setattr__`, `__delattr__`, `__getattribute__`

Comment crée-t-on une métaclasse ?

- On crée des métaclasses :
  - en créant une classe qui hérite de `type` (et non pas d'`object`)
  - en rédefinissant les méthodes que l'on veut adapter

Attention

Ne pas oublier d'appeler la méthode (surchargée) de la classe `type`

Script objets.py : une nouvelle métaclasse M

```
1  #!/usr/bin/env python
2
3  class M(type):
4      def __new__(*args):
5          print(f"Appel de __new__ de M: {args}")
6          return type.__new__(*args)
7
8      def __init__(*args):
9          print(f"Appel de __init__ de M: {args}")
10         return type.__init__(*args)
11
12     def __getattribute__(*args):
13         print(f"Appel de __getattribute__ de M: {args}")
14         return type.__getattribute__(*args)
15
16     def __getattr__(*args):
17         print(f"Appel de __getattr__ de M: {args}")
18         raise AttributeError()
19
20     def __call__(*args, **kargs):
21         print(f"Appel de __call__ de M: {args}")
22         return type.__call__(*args, **kargs)
```

ROUEN NORMANDIE

Script objets.py : une classe C instance de M

```
24 print("Création de la classe C")
25 class C(object, metaclass=M):
26     ca = 10
27
28     def __new__(*args):
29         print(f"Appel de __new__ de C: {args}")
30         return object.__new__(*args)
31
32     def __init__(self):
33         print(f"Appel de __init__ de C: {self}")
34         self.a = 1
35
36     def __getattribute__(*args):
37         print(f"Appel de __getattribute__ de C: {args}")
38         return object.__getattribute__(*args)
39
40     def __getattr__(*args):
41         print(f"Appel de __getattr__ de C: {args}")
42         return None
```

INSA ROUEN NORMANDIE

Script `objets.py` : le main

```
44 if __name__ == "__main__":
45     print("Instanciation de c")
46     c = C()
47     print("obtention de c.a")
48     print(c.a)
49     print("obtentino de c.b")
50     print(c.b)
51     print("obtention de c.ca")
52     print(c.ca)
53     print("obtention de C.ca")
54     print(C.ca)
```

Exécution du script objets.py

```
$ python objets.py
Création de la classe C
Appel de __new__ de M: (<class '__main__.M'>, 'C', (<class 'object'>,),
  {'__module__': '__main__',
  '__qualname__': 'C',
  'ca': 10,
  '__new__': <function C.__new__ at 0x7f95f46cabf8>,
  '__init__': <function C.__init__ at 0x7f95f46cac80>,
  '__getattribute__': <function C.__getattribute__ at 0x7f95f46cad08>,
  '__getattr__': <function C.__getattr__ at 0x7f95f46cad90>})

Appel de __init__ de M: (<class '__main__.C'>, 'C', (<class 'object'>,),
  {'__module__': '__main__',
  '__qualname__': 'C',
  'ca': 10,
  '__new__': <function C.__new__ at 0x7f95f46cabf8>,
  '__init__': <function C.__init__ at 0x7f95f46cac80>,
  '__getattribute__': <function C.__getattribute__ at 0x7f95f46cad08>,
  '__getattr__': <function C.__getattr__ at 0x7f95f46cad90>})
```

Exécution du script objets.py (suite)

```
Instanciation de c
Appel de __call__ de M: (<class '__main__.C'>,)
Appel de __getattribute__ de M: (<class '__main__.C'>, '__new__')
Appel de __new__ de C: (<class '__main__.C'>,)
Appel de __init__ de C: : <__main__.C object at 0x7f382c4317b8>
obtention de c.a
Appel de __getattribute__ de C: (<__main__.C object at 0x7f95f5f48860>, 'a')
1
obtentino de c.b
Appel de __getattribute__ de C: (<__main__.C object at 0x7f95f5f48860>, 'b')
Appel de __getattr__ de C: (<__main__.C object at 0x7f95f5f48860>, 'b')
None
obtention de c.ca
Appel de __getattribute__ de C: (<__main__.C object at 0x7f95f5f48860>, 'ca')
10
obtention de C.ca
Appel de __getattribute__ de M: (<class '__main__.C'>, 'ca')
10
```

ROUEN NORMANDIE

Python
Les décorateurs

Nicolas Delestre

INSA
ROUEN NORMANDIE

Précisions sur les fonctions 1 / 3

Objets retournés

- Une fonction retourne `None` (par exemple si pas de `return`) ou un objet
- Une fonction possède des variables locales qui référencent des objets
- Des invocations d'une même fonction peuvent retourner des objets différents (au sens de l'identité)

```
1 def foo1(a):
2     interne = [a]
3     return interne
```

```
>>> a1=foo1(1)
>>> a2=foo1(1)
>>> a3=foo1(2)
>>> a1
[1]
>>> a2
[1]
>>> a3
[2]
```

```
>>> a1 is a2
False
>>> a1 is a3
False
>>> a2 is a3
False
```

Précisions sur les fonctions 2 / 3

Retourner une fonction (lambda)

- Une fonction est un objet
- Une fonction (principale) peut très bien retourner une fonction
- Chaque appel de le fonction (principale) peut retourner une fonction différente (au sens de l'identité)

```
1 def foo2(a):
2     interne = lambda : a
3     return interne
```

```
>>> b1=foo2(1)
>>> b2=foo2(1)
>>> b3=foo2(2)
>>> b1()
1
>>> b2()
1
>>> b3()
2
```

```
>>> b1 is b2
False
>>> b1 is b3
False
>>> b2 is b3
False
```

Précisions sur les fonctions 3 / 3

Retourner une fonction (interne)

- Une fonction interne est une variable locale
- Elle peut donc être retournée
- Chaque appel à la fonction principale retournera une fonction différente (au sens de l'identité)

```
1 def foo3(a):
2     def interne():
3         return a
4     return interne
```

```
>>> c1=foo3(1)
>>> c2=foo3(1)
>>> c3=foo3(2)
>>> c1()
1
>>> c2()
1
>>> c3()
2
```

```
>>> c1 is c2
False
>>> c1 is c3
False
>>> c2 is c3
False
```

Les décorateurs

Rappels

- Des instructions, commençant par un `@`, se trouvant juste avant une définition d'une fonction qui modifie le comportement de cette fonction
- Exemples : `@property`, `@staticmethod`, `@fixture`, etc.

Comment ?

- Développer une fonction qui prend une fonction en paramètre et qui retourne une fonction, par exemple :

```
@dec2                                    ⇒    def func(arg1, arg2, ...):
@dec1                                             pass
def func(arg1, arg2, ...):                 func = dec2(dec1(func))
    pass
```

Exemple issue de `https://www.python.org/dev/peps/pep-0318/`

Le décorateur « cache » 1 / 4

Objectif

Sauvegarder les résultats d'une fonction : lorsque l'on appelle une deuxième fois cette fonction avec les même paramètres, il n'y a pas de nouveau le calcul du résultat

Principe

- Attacher à la fonction un dictionnaire dont les clés sont les paramètres effectifs de la fonction et les valeurs les valeurs calculées
- À chaque appel de fonction, on vérifie si les paramètres effectifs sont une clé du dictionnaire, et dans ce cas on retourne la valeur associée. Sinon on calcule la valeur et on la sauvegarde dans le dictionnaire avant de la retourner

INSA ROUEN NORMANDIE

Code du décorateur cache

```
1  def cache(fnt_a_cacher):
2      def fnt_avec_cache(*args):
3          try:
4              fnt_avec_cache.le_cache
5          except AttributeError:
6              fnt_avec_cache.le_cache = {}
7          le_cache = fnt_avec_cache.le_cache
8          if args in le_cache.keys():
9              return le_cache[args]
10         else:
11             val = fnt_a_cacher(*args)
12             le_cache[args] = val
13             return val
14     return fnt_avec_cache
```

INSA ROUEN NORMANDIE

Le décorateur « cache » 3 / 4

cnp sans utilisation du décorateur cache

```
def cnp(n,p):
    if p == 1 or n == p:
        return 1
    else:
        return cnp(n-1, p-1) + cnp(n-1, p)
```

Sous ipython3

```
In [11]: %time cnp(30,20)
CPU times: user 4.59 s, sys: 4 ms, total: 4.59 s
Wall time: 4.59 s
Out[11]: 20030010

In [12]: %time cnp(31,20)
CPU times: user 12.1 s, sys: 0 ns, total: 12.1 s
Wall time: 12.1 s
Out[12]: 54627300
```

Le décorateur « cache » 4 / 4

cnp avec utilisation du décorateur cache

```
@cache
def cnp(n,p):
    if p == 1 or n == p:
        return 1
    else:
        return cnp(n-1, p-1) + cnp(n-1, p)
```

Sous ipython3

```
In [2]: %time cnp(30,20)
CPU times: user 870 us, sys: 0 ns, total: 870 us
Wall time: 978 us
Out[2]: 20030010

In [3]: %time cnp(300,200)
CPU times: user 33.5 ms, sys: 30 us, total: 33.6 ms
Wall time: 33.3 ms
Out[3]: 2772167642172376496522255684217603720186977337162432472442022438203170
885549339080
```