UNIVERSITÉ
CÔTE D'AZUR

Programmation impérative en Python

Cours 7. Modules et types abstraits

Olivier Baldellon

Courriel : prénom.nom@univ-cotedazur.fr

Page professionnelle : https://upinfo.univ-cotedazur.fr/~obaldellon/

Licence 1 — Faculté des sciences et ingénierie de Nice — Université Côte d'Azur

Sommaire

- Un nom de fichier Python se termine par `.py` et ne contient que des lettres minuscules, des chiffres et des soulignés. Aucun espace !
  - `essai2.py`
  - `essai_tortue.py`
  - ~~`Essai tortue.py`~~
- Un module est un fichier Python `nom_du_module.py` contenant
  - des définitions ;
  - des instructions (par exemple d'affichage).
- Il y a deux types de modules.

Les scripts

Suite d'instructions destinées à être directement exécutées (avec la touche F5 par exemple)

Les bibliothèques (*library*)

Ensemble de fonctions destinées à être utilisées par un autre module.

- On peut mélanger les deux mais ce n'est pas une bonne pratique.

- On crée un fichier `tva.py`
  - avec des définitions de variables et de fonctions (`TVA`, `COEF`, `prix_ttc`)
  - et des instructions (le `print`)
  - `tva` est donc à la fois une bibliothèque et un script.
  - en général, le mélange des genres n'est pas une bonne pratique.

tva.py

```
TVA = 20                # définition d'une variable
COEF = 1 + TVA / 100 # définition d'une variable

def prix_ttc(p):        # définition d'une fonction
    global COEF
    return p * COEF

print('Module tva.py chargé !') # instruction
```

- On peut l'utiliser à partir de la console.

SHELL

```
>>> import tva # tva.py dans le répertoire de travail
Module tva.py chargé !
>>> tva.TVA
20
>>> tva.prix_ttc(10)
12.0
```

- On prend le même fichier `tva.py`

tva.py

```
TVA = 20
COEF = 1 + TVA / 100

def prix_ttc(p):
    global COEF
    return p * COEF

# print('Module tva.py chargé !')
```

- et on l'utilise alors dans un script

SCRIPT

```
import tva # tva.py est dans le même dossier

def bilan(x):
    prix = tva.prix_ttc(x)
    print(f'Avec un taux de {tva.TVA}%,', end=' ')
    print(f'le prix ttc est {prix}€.')
```

SHELL

```
>>> bilan(10)
Avec un taux de 20%, le prix ttc est 12.0€.
```

tva.py
```
TVA = 20
COEF = 1 + TVA / 100

def prix_ttc(p):
    global COEF
    return p * COEF
```

- Le module `tva` définit les variables `TVA`, `COEF` et la fonction `prix_ttc`.
- La portée de ces définitions est restreinte au module
  - elles ne sont pas visibles directement dans un autre module.
  - sauf à utiliser le préfixe `tva.` : `tva.COEF`, etc.

SHELL
```
>>> COEF=3
>>> tva.prix_ttc(10) #10*COEF?
12.0
```

SHELL
```
>>> tva.COEF = 3
>>> tva.prix_ttc(10)
30
```

- Les espaces de noms (*namespace*) permettent d'éviter les conflits de noms involontaires.
  - On peut utiliser le nom `COEF` sans conflit avec celui de `tva.py`
  - Mais si on veut vraiment modifier `tva.COEF`, on peut !

- Utilisée dans le module autre, l'instruction `import tva` :
  - exécute le fichier `tva.py`
  - rend disponibles les définitions de `tva.py` depuis le fichier `autre.py`.

tva.py
```
TVA = 20
COEF = 1 + TVA / 100

def prix_ttc(p):
    global COEF
    return p * COEF

print('Module tva.py chargé !')
```

autre.py
```
import tva #exécute tva

def bilan(x):
    prix = tva.prix_ttc(x)
    print('Taux :',tva.TVA,end='')
    print(' ; Prix ttc :', prix)

bilan(10)
```

- Il suffit de préfixer les noms du module `tva.py` par `tva.` (tva point)
  - `prix_ttc(10)` devient `tva.prix_tcc(10)`

SHELL
```
>>> import autre # autre est un script : exécute des instructions
Module tva.py chargé !
Taux : 20 ; Prix ttc : 12.0
>>> autre.bilan(100) # autre est aussi une bibliothèque
Taux : 20 ; Prix ttc : 120.0
```

- Il est préférable d'utiliser un module comme simple bibliothèque.

- Utilisée dans le module `autre`, l'instruction `from tva import TVA` :
  - Exécute le fichier `tva.py`
  - Introduit le mot `TVA` dans l'espace de noms du module `autre`
  - En pratique le mot `TVA` devient un nom de variable du module `autre`.

SHELL
```
>>> from tva import TVA
>>> TVA # variable TVA est définie & utilisable sans préfixe
20
>>> tva.TVA # nous n'avons pas importé le module tva
Traceback (most recent call last):
  File "<console>", line 1, in <module>
NameError:  name 'tva' is not defined
>>> COEF # du module tva nous n'avons importé que TVA
Traceback (most recent call last):
  File "<console>", line 1, in <module>
NameError:  name 'COEF' is not defined
```

- Pour introduire tout l'espace de noms d'un module : `from tva import *`
  - Mais c'est beaucoup plus risqué qu'un simple `import tva`
  - S'obliger à utiliser le préfixe `tva` évite les conflits.
  - Cela évite de redéfinir silencieusement une variable déjà existante.

- Il ne peut y avoir qu'une variable `TVA`

SHELL
```
>>> TVA=3
>>> from tva import TVA
>>> TVA
20
>>> from tva_réduite import TVA
>>> TVA
5
```

- Première solution

SHELL
```
>>> TVA=3
>>> import tva
>>> import tva_réduite
>>> ( TVA , tva.TVA , tva_réduite.TVA )
(3, 20, 5)
```

- Pour éviter des conflits, on peut importer une valeur sous un autre nom.

SHELL
```
>>> TVA=3
>>> from tva import TVA as TVAP          # TVAP = TVA Pleine
>>> from tva_réduite import TVA as TVAR # TVAR = TVA Réduite
>>> (TVA,TVAP,TVAR)
(3, 20, 5)
```

- Pour importer tous les noms du module `tva` en utilisant le préfixe

```shell
>>> import tva
>>> tva.COEF
1.2
```

- Pour importer une seule variable/fonction de `tva` sans préfixe
  - pour tout importer sans préfixe (à éviter) `from tva import *`

```shell
>>> from tva import COEF
>>> COEF
1.2
```

- Pour importer une seule variable/fonction de `tva` en changeant son nom

```shell
>>> from tva import COEF as coefficient
>>> coefficient
1.2
```

- Pour ceux de l'UE Syst. 1, on peut utiliser les modules comme des scripts.
- Il faut les commencer par une ligne en commentaire : le *shebang* :
  - Pour indiquer au shell Unix quel langage utiliser.
  - On utilise d'autre shebang pour d'autre langage (`#!/bin/bash`)

script.py

```
#!/usr/bin/python3

for i in range(1,4):
    print(i)
```

- Il faut aussi que votre fichier soit exécutable (commande Unix `chmod`).

SHELL

```
olivier@valrose:~ $ chmod u+x script.py
olivier@valrose:~ $ ./script.py
1
2
3
```

- On lance le script en préfixant son nom par `./`

Script principal ou bibliothèque ?

- Parfois, on veut savoir si un module est utilisé en script ou en bibliothèque.
- Astuce : le script principal est appelé `__main__` par Python.

module.py

```
def carré(x):
    return x*x

if __name__ == '__main__':
    print('> [script principal]')
else:
    print('> [bibliothèque]')
    print('>', __name__)
```

script.py

```
import module

print('Bonjour')
print(__name__)
```

SHELL

```
olivier@valrose:~ $ python3 module.py
> [script principal]
olivier@valrose:~ $ python3 script.py
> [bibliothèque]
> module
Bonjour
__main__
```

- On peut aussi appeler un script en faisant « `python3 monscript.py` »

Sommaire

- Qu'est-ce qu'un nombre ?
  - Une longueur ? comme 3 cm.
  - Une quantité d'objets ? trois carottes.
  - Un symbole ? le chiffre arabe « 3 »
  - Un ensemble ? en théorie des ensembles, on a :

$$3 = \left\{ \quad \{\} \quad \{\{\}\} \quad \{\{\}\{\{\}\}\} \quad \right\}$$

- Peu importe : Les nombres se définissent par leur propriétés.
  - on peut leur appliquer des opérations ($+$, $-$, $\times$, $\div$)
  - on peut les ordonner ($\leq$ ou $\geq$)
  - ils vérifient certaines propriétés (distributivité, ensemble infini, etc.)

- Et ceci, quelle que soit la façon dont on les construit mathématiquement.

- Nous avons rencontré différents types : flottants, booléens, listes, tuples…
- Nous voulons travailler maintenant avec des matrices.
  - Mais notre langage ne les a pas prévues (il ne peut pas tout prévoir).
  - Il y a plusieurs façons de définir une matrice (liste, tuple, etc.)
- Nous allons travailler en deux temps :
  - Nous allons créer dans un module un nouveau type abstrait matrice
    - On choisit une façon de définir les matrices.
    - On écrit une collection de fonctions (création, opérations, affichage).
  - Nous allons ensuite utiliser le type abstrait.
    - Nous oublions comment sont codées les matrices.
    - Nous travaillons uniquement avec les fonctions du module.

Type abstrait : matrice 2 × 2

- Matrice 2×2 : 4 nombres (2 lignes, 2 colonnes)
  - Il faut savoir construire une matrice
  - Il faut savoir accéder à ses éléments

$$\begin{pmatrix} m_{0,0} & m_{0,1} \\ m_{1,0} & m_{1,1} \end{pmatrix}$$

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

- Les matrices définies comme des tuples

SCRIPT

```
def matrice(a,b,c,d):
    return (a, b, c, d)

def accès(M, li, co): # m_{i,j}
    return M[2 * li + co]
```

SHELL

```
>>> A=matrice(1,2,3,4)
>>> accès(A,1,0)
3
>>> A #comment est construit A
(1, 2, 3, 4)
```

- Les matrices comme listes de listes

SCRIPT

```
def matrice(a,b,c,d):
    return [[a,b],[c,d]]

def accès(M, li, co): # m_{i,j}
    return M[li][co]
```

SHELL

```
>>> A=matrice(1,2,3,4)
>>> accès(A,1,0)
3
>>> A #comment est construit A
[[1, 2], [3, 4]]
```

- Conclusion :
  - Il n'y a pas une unique représentation concrète des matrices abstraites.
  - Le concept de matrice est défini par un ensemble de fonctions
    - ici `matrice(…)` et `accès(…)`
  - Si je veux définir de nouvelles fonctions ; je n'utilise que ces deux-là.

SCRIPT

```
import matrice

def trace(M):
    a=matrice.accès(M, 0, 0)
    d=matrice.accès(M, 1, 1)
    return a+d

def déterminant(M):
    a=matrice.accès(M, 0, 0)
    b=matrice.accès(M, 0, 1)
    c=matrice.accès(M, 1, 0)
    d=matrice.accès(M, 1, 1)
    return a*d - b*c
```

matrice.py

```
def matrice(a,b,c,d):
    return [[a,b],[c,d]]

def accès(M, li, co): # m_{i,j}
    return M[li][co]
```

2 fichiers possibles.

matrice.py

```
def matrice(a,b,c,d):
    return (a, b, c, d)

def accès(M, li, co): # m_{i,j}
    return M[2 * li + co]
```

- En dehors du module, on ignore la représentation concrète des matrices
- on en fait abstraction.

- Il y a deux acteurs dans notre histoire :
  - Le développeur qui écrit le module
  - L'utilisateur qui fait appel aux fonctions du module.
  - Entre les deux il y a le type abstrait (ici matrice 2×2)
- L'interface définit un type abstrait
  - Elle doit être documentée.
  - C'est le nom des fonctions et ce qu'elles font (sémantiques)
  - Elle ne doit jamais changer !
- Le développeur fait le choix de la structure.
  - Il écrit les fonctions de l'interface
  - Il peut modifier la structure (pour gagner en efficacité par exemple)
  - Il ne doit pas toucher à l'interface.
- L'utilisateur écrit des algorithmes/programmes en utilisant l'interface
  - Une documentation suffit, avec la complexité des fonctions.
  - Il n'a pas à se soucier des changements fais par le développeur
  - Il ne doit pas utiliser la structure interne.

- Python n'offre pas les nombres rationnels exacts (fractions).
  - Créons un type abstrait correspondant !
- Mathématiquement on représente un rationnel par une fraction $\frac{p}{q}$
  - avec $q > 0$
  - et $\text{pgcd}(p, q) = 1$ (la fraction est irréductible)
- Informatiquement, on choisit de les représenter par un couple d'entier
  - le numérateur p et le dénominateur q,
  - les mêmes conditions mathématiques doivent être vérifiées.
- Que mettre dans notre module ?
  - Les fonctions qui ont besoin d'utiliser la structure interne
  - Ces fonctions formeront l'interface
  - Les autres fonctions n'utiliseront que l'interface indépendamment de la structure interne

rationnel.py

```
import math

# pour créer le rationnel p/q avec p et q entiers
def rationnel(p, q):
    if q == 0:
        raise ValueError('dénominateur nul !')
    if q < 0:
        (p, q) = (-p, -q)
    g = math.gcd(p, q) # ici, q est forcément > 0
    return (p//g, q//g) # on simplifie la fraction

def numérateur(r):
    return r[0]

def dénominateur(r):
    return r[1]

def représ(r): # la représentation externe de r
    if r[0] == 0: return '0'
    if r[1] == 1: return str(r[0])
    return f'{r[0]}/{r[1]}'
```

- Définissons maintenant l'addition sur les rationnels.

SCRIPT
```
import rationnel

def addition_rationnel(a,b):
    pa = rationnel.numérateur(a)
    qa = rationnel.dénominateur(a)
    pb = rationnel.numérateur(b)
    qb = rationnel.dénominateur(b)
    return rationnel.rationnel(pa*qb + qa*pb,qa*qb)
```

- On teste notre fonction

SHELL
```
>>> from rationnel import *
>>> r1 = rationnel(6, -4)
>>> r2 = rationnel(1, 3)
>>> r3 = addition_rationnel(r1, r2)
>>> print(f'{représ(r1)} + {représ(r2)} = {représ(r3)}')
-3/2 + 1/3 = -7/6
```

- Nul besoin de se soucier de la simplication des fractions !

- Malheureusement on a toujours accès à la structure interne.

SHELL
```
>>> print(r1)
(-3, 2)
>>> print(représ(r1))
-3/2
```

- Python permet de faire des choses plus propres
  - écrire `r1 + r2` directement (au lieu d'une fonction `addition`)
  - rendre invisible la structure interne de `r1`
  - faire en sorte que `print(r1)` utilise la fonction `représ`
  - définir un nouveau type `rationnel`

- Mais Python utilise pour cela le formalisme de la programmation objet
  - C'est l'objet de la partie suivante (la méthode utilisée sera plutôt classe)

- Les principes vus restent valides quel que soit le langage
  - La programmation objet n'est qu'une généralisation de ces principes
  - Les autres langages (objet ou non) ont tous des mécanismes similaires

Sommaire

- Créons une classe `Rationnel` pour que le type soit intégré à Python.

rationnels.py

```
import math

class Rationnel:
    def __init__(self,p,q):
        if q == 0:
            raise ValueError('dénominateur nul !')
        elif q < 0:
            (p, q) = (-p, -q)
        g = math.gcd(p, q) # ici, q est forcément > 0
        self.numérateur=p//g
        self.dénominateur=q//g # on simplifie la fraction
```

SHELL

```
>>> from rationnels import *
>>> a = Rationnel(10,20)
>>> a
<rationnels.Rationnel object at 0x7f443b28ced0>
>>> print(a)
<rationnels.Rationnel object at 0x7f443b28ced0>
```

rationnels.py

```
class Rationnel:
    def __init__(self,p,q): #...

    def __repr__(self):
        if self.dénominateur == 1:
            return str(self.numérateur)
        else:
            p=self.numérateur
            q=self.dénominateur
            return f'{p}/{q}'
```

SHELL

```
>>> a = Rationnel(10,20)
>>> a
1/2
>>> print(a)
1/2
>>> type(a)
<class 'rationnels.Rationnel'>
```

rationnels.py

```
class Rationnel:
    def __init__(self,p,q): #...

    def __repr__(self): #...

    def __getitem__(self,i):
        if i==0:
            return self.numérateur
        elif i==1:
            return self.dénominateur
        else:
            raise IndexError("L'indice doit être 0 ou 1")
```

SHELL

```
>>> (a[0],a[1]) # Rappel : a = Rationnel(10,20)
(1, 2)
>>> a[2] # a.__getitem__(2)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "rationnels.py", line 31, in __getitem__
    raise IndexError("L'indice doit être 0 ou 1")
IndexError:  L'indice doit être 0 ou 1
>>> a[0]=3
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError:  'Rationnel' object does not support item
assignment
```

rationnels.py

```python
class Rationnel:
    def __init__(self,p,q): # ...

    def __repr__(self): # ...

    def __getitem__(self,i): # ...

    def __add__(self,r):
        if type(r)==float:
            return self[0]/self[1]+r
        if type(r)==int:
            r = Rationnel(r,1)
        d  = self[1] * r[1]
        n1 = self[0] * r[1]
        n2 = r[0] * self[1]
        return Rationnel(n1+n2,d)
```

SHELL

```
>>> (a,b,c,d) =(Rationnel(1,2), Rationnel(3,4), 1, 1.0)
>>> print(f"{a} + {b} = {a+b}") # ratio. + ratio. = ratio.
1/2 + 3/4 = 5/4
>>> print(f"{a} + {c} = {a+c}") # ratio. +  int   = ratio.
1/2 + 1 = 3/2
>>> print(f"{a} + {d} = {a+d}") # ratio. + float  = float
1/2 + 1.0 = 1.5
```

- `a+1` est remplacé par `a.__add__(1)`
- `1+a` est remplacé par défaut par `1.__add__(a)`
  - non définie pour `a` rationnel.
- Pour définir l'addition à droite, il faut une nouvelle méthode.

rationnels.py

```
class Rationnel:
    def __init__(self,p,q): # ...

    def __repr__(self): # ...

    def __getitem__(self,i): # ...

    def __add__(self,r): # ...

    def __radd__(self,r):
        return self.__add__(r)
```

SHELL

```
>>> a = Rationnel(3,4)
>>> a+1 # remplacé par a.__add__(1)
7/4
>>> 1+a # remplacé par a.__radd__(1)
7/4
```

Voici les principales possibilités offertes par Python.

- Attention : chaque nom de méthode commence et se termine par `__`
- Pour les curieux, qui veulent allez plus loin (à vous de chercher le détail).
- Il en existe encore beaucoup d'autres

- Pour l'affichage :
  - `repr` (affichage dans le toplevel),
  - `str` (affichage avec `print`)

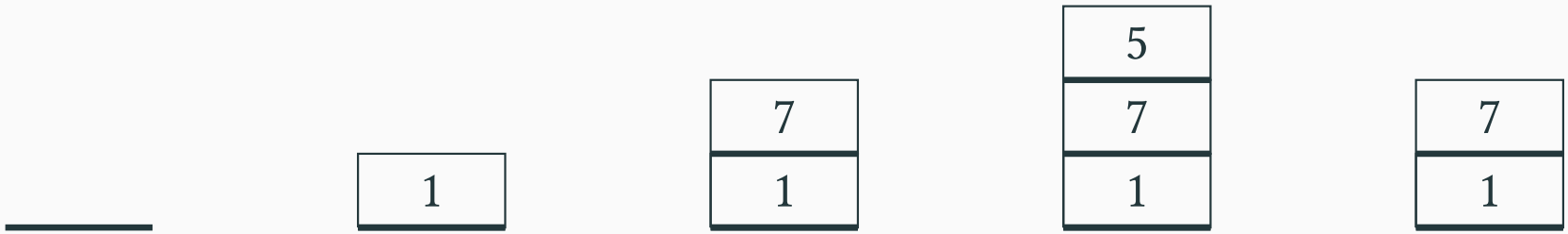
- Les opérations :
  - `add` (`+`), `sub` (`-`), `mul` (`*`), `floordiv` (`/`), `truediv` (`//`), `mod` (`%`), `pow` (`**`)
  - Pour définir `b+a` où `b` n'est pas un objet du même type
  - `radd` (`+`), `rsub` (`-`), ... , `rpow` (`**`)

- Les comparaisons :
  - `lt` (`<`), `le` (`<=`), `eq` (`==`), `ne` (`!=`), `gt` (`>`), `ge` (`>=`)

- Pour créer ses propres séquences :
  - `getitem` (`a[i]`), `setitem` (`a[i]=x`), `len` (`len(a)`)
  - `iter`, `next` pour la syntaxe `for e in a`

Sommaire

- La pile est un type de structure, comme le tuple ou la liste, très utilisé en informatique.



- Fonctionnement :
  - Au début la pile est vide
  - On empile 1
  - On empile 7
  - On empile 5
  - On dépile 5
  - À la fin le sommet vaut 7

Implémentation du type abstrait

- On utilise une liste comme structure interne

pile.py

```
PileErreur=ValueError('Pile vide')

def nouvelle_pile():
    return []

def empile(L,e):
    L.append(e)

def dépile(L):
    if L == []: raise PileErreur
    return L.pop()

def sommet(L):
    if L == []: raise PileErreur
    return L[len(L)-1]

def est_vide():
    return L==[]
```

SHELL

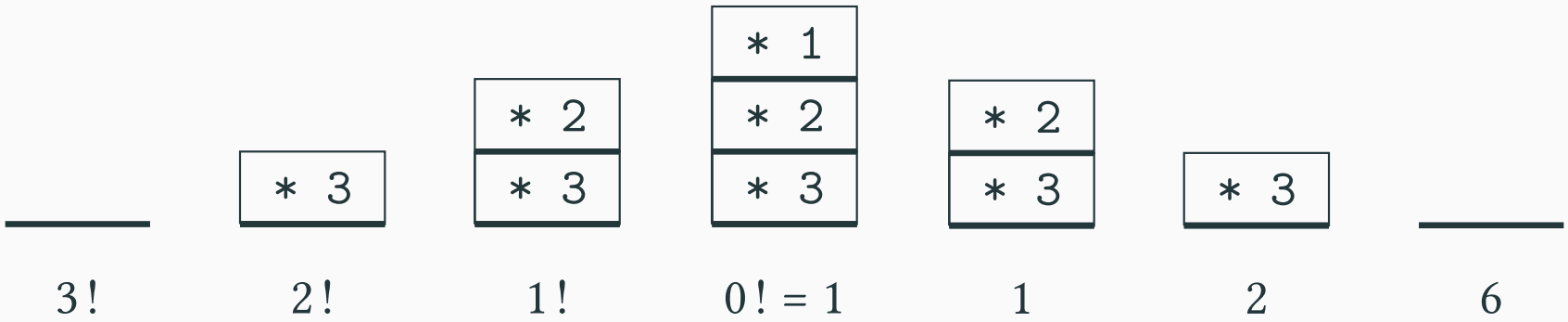
```
>>> import pile
>>> P = pile.nouvelle_pile()
>>> pile.empile(P,3)
>>> pile.sommet(P)
3
>>> pile.dépile(P)
3
>>> pile.sommet(P)
Traceback (most recent call last):
  File "<console>", line 1, in
<module>
  File "pile.py", line 13, in
sommet
    if L == []: raise PileErreur
                ^^^^^^^^^^^^^^^^

ValueError:  Pile vide
```

À quoi servent les piles ?

- Les piles sont souvent utiles pour stocker des calculs intermédiaires.
- Typiquement pour le calcul de la factorielle.
  - Il faut calculer `fact(n-1)` avant de faire le `* n`
  - On met les calculs « `* n` » dans la pile.
  - et dès qu'on tombe sur `0!` on dépile et applique les calculs

SCRIPT

```
def fac(n):
    if n == 0:
        return 1
    else:
        return fac(n - 1) * n
```

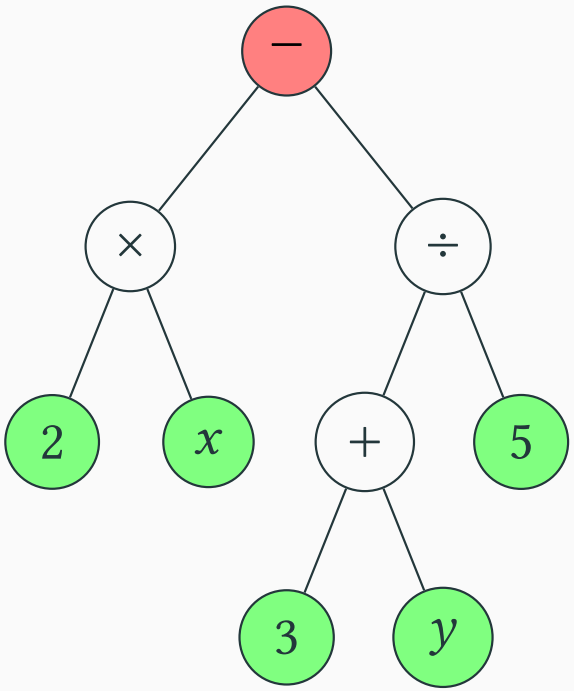


- Le véritable mécanisme fonctionne aussi avec des piles : cours 5 partie v

Sommaire

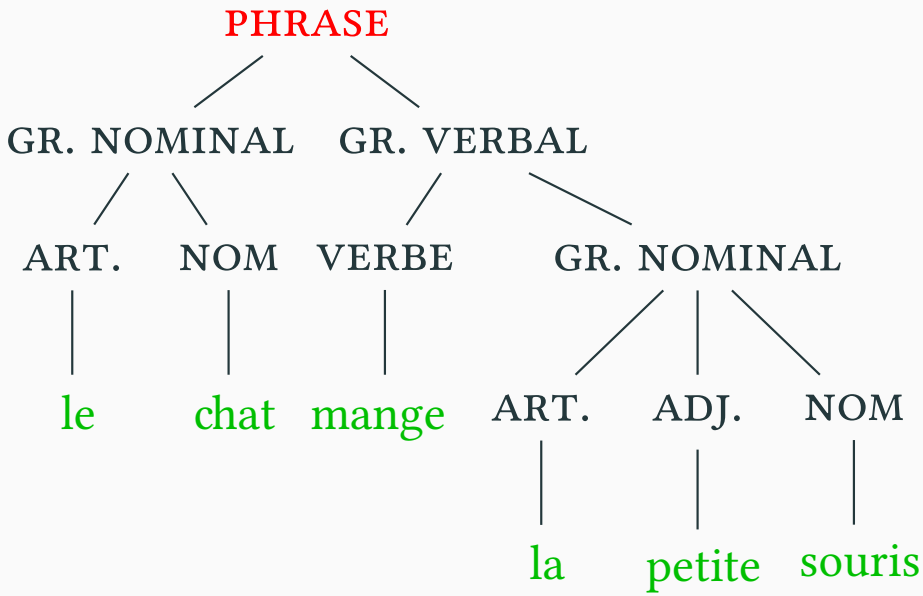
- Un arbre (graphe acyclique) est un objet fondamental en informatique.
  - Un arbre a une racine (en rouge)
  - Un arbre a des nœuds (avec des enfants) ; la racine est un nœud
  - Un arbre a des feuilles (sans enfant) (en vert)

Arbre binaire d'expression



1 racine, 4 nœuds, 5 feuilles

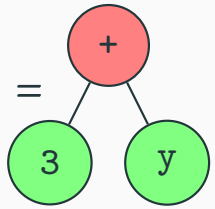
Un arbre grammatical



1 racine, 10 nœuds, 6 feuilles

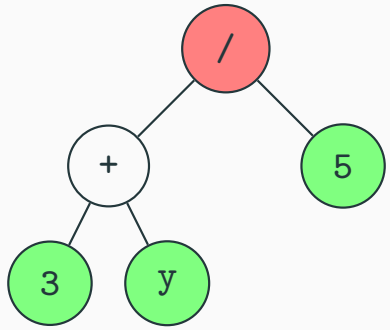
Les arbres binaires d'expression : principe

- Il y a deux types d'arbres :
  - Les arbres simples correspondant à des feuilles
  - Les arbres composés constitués d'une racine et deux sous-arbres
- On choisit une opération r, un arbre composé A_1, un arbre simple A_2 :

$r = /$ $\quad A_1 =$ 

$\quad A_2 =$ 

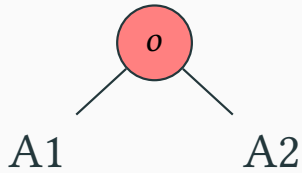
- On peut construire un nouvel arbre composé :



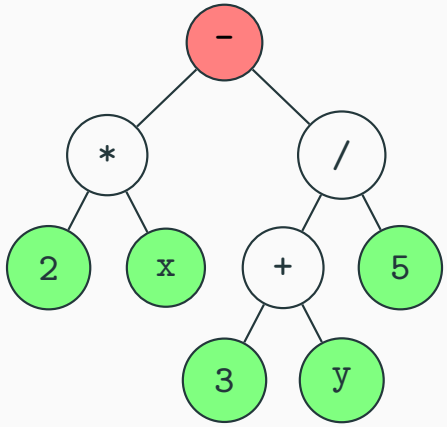
- Chaque arbre correspond à une expression : ici l'expression est `(3+y)/5`

Les arbres binaires d'expression : définitions

- On définit un arbre binaire d'expression (ABE) par récurrence :
- Cas de base : une feuille contenant un nombre ou une lettre est un ABE.
  - L'expression correspondante est le contenu de la feuille.
- Cas composé : L'arbre suivant est une ABE si et seulement si :
  - A_1 et A_2 sont des ABE
  - o une operation (`+`, `-`, `*`, `/`)
  - L'expression associée s'obtient en appliquant o aux expressions associées à A_1 et A_2.



- Exemple : l'arbre suivant correspond au calcul `(2*x)-((3+y)/5)`



- Un arbre est soit un tuple `(r,Ag,Ad)` soit une feuille (int ou str)

abe.py

```
def arbre(r, Ag, Ad):
    return (r, Ag, Ad)

def est_feuille(obj):
    return type(obj) == int or type(obj) == str

def racine(A): # A doit être un nœud
    if est_feuille(A):
        raise ValueError("une feuille n'a pas de racine")
    return A[0]

def fg(A): # A doit être un nœud
    if est_feuille(A):
        raise ValueError("une feuille n'a pas de fils")
    return A[1]

def fd(A): # A doit être un nœud
    if est_feuille(A):
        raise ValueError("une feuille n'a pas de fils")
    return A[2]
```

- Pour construire un arbre, on peut :
  - soit construire directement le tuple : c'est très mal !

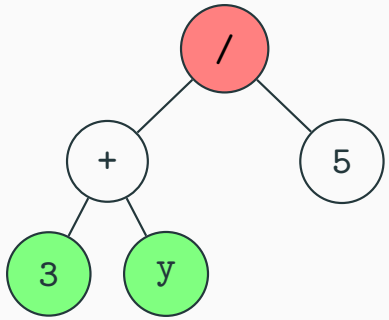
```
A = ('/', ('+', 3, 'y'), 5)
```

  - soit passer par le constructeur du type abstrait (c'est la méthode propre)

```
A = arbre( '/', arbre('+', 3, 'y'), 5)
```

SHELL

```
>>> from abe import *
>>> A = arbre('/', arbre('+', 3,'y'), 5)
>>> A
('/', ('+', 3, 'y'), 5)
>>> fg(A)
('+', 3, 'y')
>>> fd(fg(A))
'y'
>>> racine(fg(A))
'+'
```

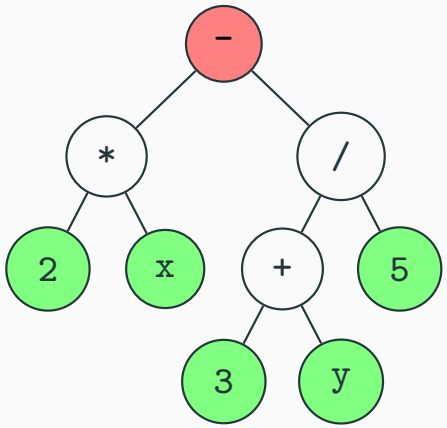


Algorithmes simples : Hauteur d'un arbre

- C'est la plus grande longueur reliant la racine à une feuille

SCRIPT

```
def hauteur(A):
    if est_feuille(A):
        return 0
    else:
        hgauche = hauteur(fg(A))
        hdroite = hauteur(fd(A))
        return 1+max(hgauche, hdroite)
```



SHELL

```
>>> A
('-', ('*', 2, 'x'), ('/', ('+', 3, 'y'), 5))
>>> ( hauteur(A) , hauteur(fg(A)) , hauteur(fd(A)))
(3, 1, 2)
```

- Notez la récurrence double (`hauteur` est appelée deux fois)
- Un arbre de hauteur h contient au plus 2^h feuilles (*exercice : pourquoi ?*).
- Réciproquement, pour un arbre de n feuilles, on s'attend à ce qu'il ait une hauteur h telle que $n \approx 2^h$, d'où $h \approx \log_2(n)$.
  - Dans le cas où $n = 2^h$ on parle d'arbre binaire complet.

- On cherche à savoir si une feuille est présente dans un arbre.

SCRIPT

```
def présente(f, A):
    if est_feuille(A):
        return f == A
    else:
        return présente(f, fg(A)) or présente(f, fd(A))
```

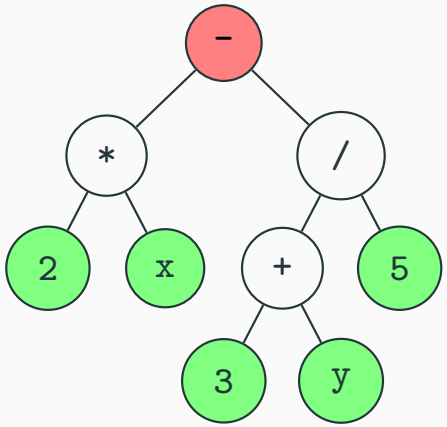
- Rappel : le `or` est paresseux.
  - On cherche d'abord dans le sous-arbre de gauche
  - Et seulement si on n'a pas trouvé, on cherche alors à droite

- Le parcours en profondeur consiste à :
  - explorer entièrement le sous-arbre de gauche avant celui de droite.
  - répéter cet ordre d'exploration dans l'exploration des sous-arbres.

- Calculer le feuillage d'un arbre revient à produire la liste plate de ses feuilles par un parcours en profondeur :

SCRIPT
```
def feuillage(A):
    if est_feuille(A):
        return [A]
    else:
        return feuillage(fg(A)) + feuillage(fd(A))
```

SHELL
```
>>> feuillage(A)
[2, 'x', 3, 'y', 5]
>>> feuillage(fd(A))
[3, 'y', 5]
```



- Exercice : Que vaut `feuillage(fd(fg(A)))` ?
- Exercice : Donner le programme qui renvoie le nombre de feuilles.

Sommaire

Valeur d'un arbre arithmétique

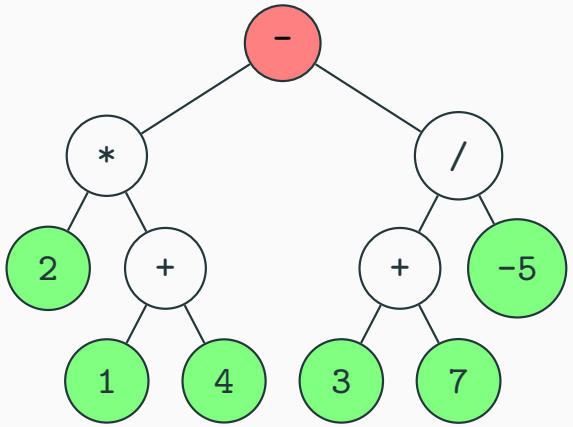
- Un arbre binaire d'expression est dit :
  - arithmétique si toutes ses feuilles sont des constantes ;
  - algébrique si au moins une feuille est une variable.

SCRIPT

```
def valeur(A): # l'arbre A doit être arithmétique
    if est_feuille(A):
        return A
    else:
        r = racine(A)
        vg = valeur(fg(A))
        vd = valeur(fd(A))
        if r == '+' : return vg + vd
        if r == '-' : return vg - vd
        if r == '*' : return vg * vd
        if r == '/' : return vg / vd
```

SHELL

```
>>> valeur(fg(A))
10
>>> valeur(A)
12.0
```



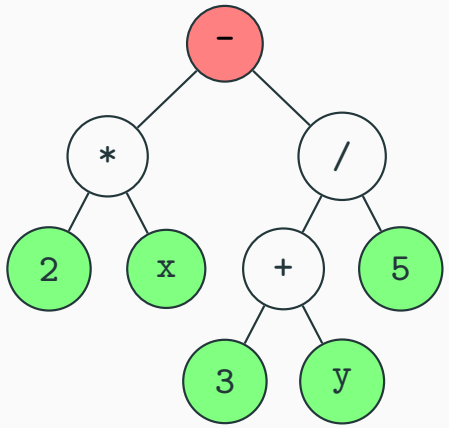
- On souhaite afficher tous les symboles sur l'arbre (nœud et feuille)
  - pour cela on va faire un parcours en profondeur préfixe.
  - D'abord on affiche la racine de l'arbre
  - On affiche récursivement les symboles du sous-arbre gauche
  - On affiche récursivement les symboles du sous-arbre droit

SCRIPT

```
def pp_préfixe(A):
    if est_feuille(A):
        print(A,end=' ')
    else:
        print(racine(A),end=' ')
        pp_prefixe(fg(A))
        pp_prefixe(fd(A))
```

SHELL

```
>>> A
('-', ('*', 2, 'x'), ('/', ('+', 3, 'y'), 5))
>>> pp_préfixe(A) ; print(' ')
- * 2 x / + 3 y 5
```



  - On remarque que l'ordre est le même que dans l'écriture de l'arbre
  - Exercice (facile) : faire une fonction qui renvoie la liste des symboles
  - Exercice inverse (TD 7) : créer l'arbre à partir de la liste des symboles

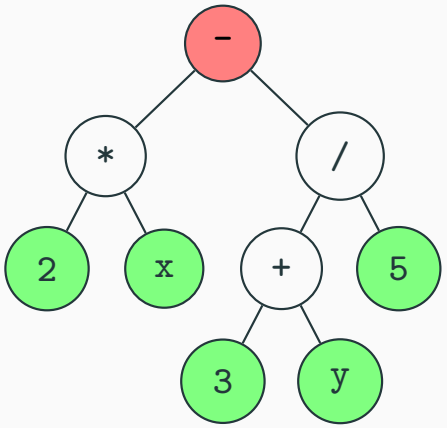
- Les arbres formant un type récursif,
  - il est naturel de programmer récursivement pour les manipuler.
- Mais on peut aussi produire un algorithme itératif (avec une boucle).
- On utilise une pile pour stocker les branches qu'il nous reste à visiter.
  - Quand on tombe sur un sous-arbre à visiter, à l'ajoute à la pile.
  - On traite les sous-arbres du haut de la pile en premier
- Exemple : calcul du nombre de feuilles de A en itératif.
  - Initialisation : je mets l'arbre dans une pile vide
  - À chaque tour de boucle je mets le sommet de la pile dans la variable `A`
  - si `A` est un arbre composé : il me reste à visiter les deux fils
    - j'empile le fils droit de `A`
    - j'empile le fils gauche de `A`
  - si `A` est un arbre simple (une feuille)
    - J'ajoute `1` au compteur de feuille

SCRIPT

```
def nb_feuilles(A): # le nombre de feuilles
    P = nouvelle_pile() ; empile(P, A)
    res = 0
    # tant qu'il reste des sous-arbres à visiter
    while not est_vide(P):
        print(P)
        A = dépile(P)
        if est_feuille(A):
            res = res + 1
        else:
            empile(P, fd(A)); empile(P, fg(A))
    return res
```

SHELL

```
>>> nb_feuilles(A)
[('-', ('*', 2, 'x'), ('/', ('+', 3, 'y'), 5))]
[('/', ('+', 3, 'y'), 5), ('*', 2, 'x')]
[('/', ('+', 3, 'y'), 5), 'x', 2]
[('/', ('+', 3, 'y'), 5), 'x']
[('/', ('+', 3, 'y'), 5)]
[5, ('+', 3, 'y')]
[5, 'y', 3]
[5, 'y']
[5]
5
```



Merci pour votre attention
Questions

?

Cours 7 — Modules et types abstraits