M1 ISD

Programmation en Python

Sylvain Conchon

Calendrier

Séances de cours :

7/09	13h30 - 17h00
9/09	9h00 - 12h30
14/09	9h30 - 12h30
16/09	9h00 - 12h30
21/09	9h30 - 12h30
23/10	9h00 - 12h30
17/10	9h00 - 11h00 (**Examen**)

Thèmes abordés :

- Les bases de Python (types de base, variables, comparaisons, boucles, conteneurs, fonctions, exceptions, etc.)
- Entrées-sorties, modules (sys, math, random, datetime, re, etc.)
- Programmation fonctionnelle (lambdas, itérateurs, décorateurs)
- Programmation orientée objets (classe, héritage)
- Calcul matriciel (pandas), tracé de courbes (matplotlib)

Évaluation

Session 1 : $CC \times 50\% + CCTP \times 50\%$

Session 2 : $EO \times 100\%$

Quelques ressources en ligne

La documentation officielle de Python 3.

`https://docs.python.org/fr/3/`

Apprendre à programmer avec Python 3. Gérard Swinnen.

`https://inforef.be/swi/download/apprendre_python3_5.pdf`

Cours de Python. Introduction à la programmation Python pour la biologie. Patrick Fuchs et Pierre Poulain.

`https://python.sdv.univ-paris-diderot.fr/cours-python.pdf`

Introduction à Python 3. Bob Cordeau.

`http://www.info.univ-angers.fr/pub/gh/tuteurs/cours-python3.pdf`

LES BASES DE PYTHON

Historique

Langage inventé par Guido Van Rossum

- Première release en 1991
- Python 1.0 en 1994
- Python 2.0 en 2000
- Python 3.0 en 2008

Développé initialement comme un langage de script (interpréteur de commandes) pour Amoeba (OS)

Depuis 2020, Python 2 n'est plus supporté. La version actuelle est Python 3.9

Généralités

Avantages :

Python est un langage de programmation généraliste (traitement de données, interfaces graphiques, réseau, jeux, calcul scientifique, intelligence artificielle, . . .)

C'est un langage interprété : la commande `python` permet d'exécuter des scripts Python

C'est un langage libre et open-source, très portable et disposant d'une *bibliothèque* très complète

Inconvénients :

Du fait qu'il est interprété, les programmes Python sont plutôt lents

Certains choix de design du langage sont limités (typage dynamique, fragilité de la syntaxe,. . .)

Un premier programme

Nous allons écrire un premier programme Python qui calcule l'âge d'une personne en 2048

Pour cela, on va simplement écrire notre programme dans un fichier `age.py`, où `.py` est l'extension des programmes Python

Un premier programme : age.py

```
n = input('Quelle est votre annee de naissance ? ')

annee = int(n)

age = 2048 - annee

print('Vous aurez',age,'ans en 2048')
```

L'interpréteur Python

Pour exécuter un programme Python, il suffit de lancer l'interpréteur Python dans un terminal, en lui donnant comme argument le nom du programme à exécuter

```
> python3.9 age.py
Quelle est votre annee de naissance ? 1999
Vous aurez 49 ans en 2048
```

Forme générale d'un programme

Un programme Python est une suite d'instructions ou de déclarations (de variable ou fonction) qui sont exécutées dans l'ordre, de haut en bas du fichier

Contrairement à certaines langages, il n'y a pas de fonction principale dans un programme Python (comme `main`)

La boucle d'interaction

L'interpréteur Python possède un mode interactif qui permet d'évaluer des instructions, comme dans un shell. Il est très pratique pour tester des petits morceaux de programme.

Pour entrer dans ce mode, il suffit de lancer l'interpréteur dans un terminal, sans lui donner de nom de fichier à exécuter

```
> python3.9
Python 3.9.6 (v3.9.6:db3ff76da1, Jun 28 2021, 11:49:53)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> 40 + 2
42
>>> print('Hello')
Hello
>>> x = 42
>>> x + 10
52
>>>
```

Les trois chevrons >>> constituent l'invite de commandes.

On peut quitter ce mode en tapant `CTRL-d`

Variables globales

Comme tous les langages de programmation, Python permet de manipuler des variables

La déclaration d'une variable se fait au même moment que son initialisation. Pour cela, il suffit simplement d'utiliser l'opérateur d'affectation = de la manière suivante :

```
var = expression
```

qui a pour effet d'ajouter une variable `var` à l'environnement des variables globales, ou de remplacer l'ancienne valeur associée à `var` dans cette table, si cette variable était déjà initialisée

La lecture du contenu d'une variable se fait classiquement à l'aide de son nom dans n'importe quelle expression

```
x + 42
```

Variables globales : nommage

Le nom d'une variable peut être constitué de lettres (en majuscule ou minuscule), de chiffres ou du caractère souligné _

Le nom ne peut pas commencer par un chiffre et il est fortement déconseillé de le faire commencer par le caractère souligné (très utilisé par Python pour nommer des variables internes au langage)

Les noms de variables sont sensibles à la casse (par ex. les noms age et `Age` représentent deux variables différentes)

Les noms de variables qui correspondent à des mots-clés de Python sont rejetés

```
>>> if = 34
 File "<stdin>", line 1
 if = 34
    ^
 SyntaxError: invalid syntax
```

Par contre, il est possible d'utiliser des noms "réservés" par Python

```
>>> print = 42
```

Ce qui est évidemment fortement déconseillé

Variables globales : spécificité Python

Contrairement à certains langages, il n'y a pas de déclaration préalable de variable en Python, en particulier il n'est pas nécessaire de spécifier le type d'une variable

Une variable peut contenir des valeurs de n'importe quel type

```
>>> x = 40 + 2
>>> x = 'Hello'
```

Python émet une erreur quand on tente de lire une variable qui n'est pas initialisée

```
>>> 42 + y
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'y' is not defined
```

Les entiers (type `int`)

En Python, les nombres entiers peuvent être de taille arbitraire (ce qui n'est pas le cas dans de nombreux langages de programmation)

Ces entiers ne sont limités que par la mémoire disponible pour les stocker

Ils peuvent représenter des nombres positifs ou négatifs

```
>>> 99999999988888888883333333333333333337777777799999999999999911111999456
99999999988888888883333333333333333337777777799999999999999911111999456
>>> -5467
-5467
```

Pour plus de lisibilité, il est possible de séparer les chiffres d'un entier en utilisant le caractère souligné _

```
>> 1_000_000 + 5
1000005
```

Les entiers : opérations

Symbole	Description
+	addition
-	soustraction
*	multiplication
/	division exacte (toujours un résultat à virgule)
//	division entière (résultat entier)
%	modulo (reste de la division entière)
**	puissance

```
>>> 1 + 1
2
>>> 2 + 3 * (4 - 6)
-4
>>> 5 / 3
1.6666666666666667
>>> 5 // 3
1
>>> 5 % 3
2
>>> 4**3
64
>>> 1 / 0
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

Les nombres à virgule (type `float`)

En Python, les nombres décimaux (ou à virgule) ont une précision limitée (ceci tient à leur encodage très compact sur 32 ou 64 bits selon la norme IEEE 754)

On peut les représenter en utilisant l'écriture scientifique

```
>>> 1.5
1.5
>>> -12.3423e13
-123423000000000.0
>>> 1.5555555555555555555555555
1.5555555555555556
>>>
```

Remarque :
$-12.3423e13 = -12.3423 \times 10^{13} = -123423000000000.0$

Attention : calculer avec des nombres flottants peut provoquer des erreurs d'arrondi

Les flottants : opérations

Les nombres flottants ont les mêmes opérations que les entiers

```
>>> 1.5 + 1.5
3.0
>>> 3.141592653589793 * 2
6.283185307179586
>>> 10.5 / 3
3.5
>>> 1.2 + 1.2 + 1.2
3.5999999999999996
>>> 4.5 ** 100
2.0953249170398634e+65
>>> 4.2 // 1.3
3.0
>>>  1.0 / 0
Traceback (most recent call last):
File "", line 1, in
ZeroDivisionError: float division by zero
```

Les booléens

Les booléens sont représentés par les deux valeurs `True` et `False`

Les opérations sur ces valeurs sont la négation `not`, le ou logique `or`, le et logique `and`

```
>>> True
True
>>> False
False
>>> not (True)
False
>>> True or False
True
>>> True and False
False
```

Les opérateurs `and` et `or` sont paresseux : ils évaluent d'abord le membre gauche, puis ensuite le membre droit, seulement si cela peut changer la valeur de l'expression

Opérateurs de comparaison

Les booléens servent à exprimer le résultat d'un test. Un cas particulier de test sont les comparaisons. Les opérateurs de comparaisons en Python sont :

Symbole	Description
==	égal
!=	différent
<	inférieur
>	supérieur
<=	inférieur ou égal
>=	supérieur ou égal

Le résultat d'une comparaison `e1 op e2` est toujours un booléen

Les chaînes de caractères (type `str`)

Les chaînes se définissent entre deux apostrophes (') ou deux guillemets (")

```
>>> print('Hello world !')
Hello world !
>>> print("J'adore Python")
J'adore Python
```

Symbole	Description
`+`	concaténation
`*`	répétition
`len(s)`	nombre de caractères
`s[i]`	accès au ième caractère (renvoie une chaîne de longueur 1)
`s[i:j]`	extraction d'une tranche (renvoie toujours une chaîne)

Les chaînes de caractères : opérations

```
>>> s1 = 'Hello'
>>> s1[2]  # le premier caractere est a l'indide 0
'l'
>>> s1[-1] # le dernier element
'o'
>>> len(s1)
5
>>> s2 = s1 + ' World!'
>>> s2
'Hello World'
>>> s3 = 3 * 'abc'
>>> s3
'abcabcabc'
>>> 'Wo' in s2
True
>>> s2[4:8]
'o Wo'
>>> s1[2:]
'llo'
```

Remarque : les chaînes de caractères sont non modifiables

Les chaînes de caractères : séquences d'échappement

Que se passe-t-il si on veut insérer un caractère ' (apostrophe) dans une chaîne ?

```
>>> 'C'est moi'
  File "<stdin>", line 1
    'C'est moi'
       ^
SyntaxError: invalid syntax
```

On doit indiquer que l'apostrophe ne marque pas la fin de la chaîne en utilisant une séquence d'échappement

```
>>> 'C\'est moi'
"C'est moi"
```

Les chaînes de caractères : autres séquences d'échappement

Comment saisir un caractère \ dans une chaîne ?

```
>>> 'Caractere antislash: \'
SyntaxError: EOL while scanning string literal
```

Il faut aussi échapper le caractère \

```
>>> 'Caractere antislash: \\'
'Caractere antislash: \\'
>>> print('Caractere antislash: \\')
Caractere antislash: \
```

Il existe d'autres séquences d'échappement : \n (retour à la ligne), \uxxxx (code Unicode en base 16), etc.

Les chaînes de caractères sur plusieurs lignes

Les chaînes de caractères qui commencent (et se terminent) par un triple apostrophe ’ ou un triple guillement " permettent le retour à la ligne dans la chaîne

```
>>> s = """Voici une
chaine sur
plusieurs lignes"""

>>> s
'Voici une\nchaine sur\nplusieurs lignes'
```

Conversions de types (1)

En Python, on peut facilement passer des types `int`, `float` et `str` à l'aide des fonctions de conversion suivantes :

Un appel `int(x)` convertit la chaîne de caractères ou le flottant `x` en un entier :

```
>>> int('145')
145
>>> int(4.5)
4
>>> int('abc')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '
                                abc'
```

Conversions de types (2)

Un appel `float(x)` convertit la chaîne de caractères ou l'entier `x` en un flottant :

```
>>> float('4.5')
4.5
>>> float(1)
1.0
>>> float(9959896598128799 9)
9.959896598128e+16
```

Un appel `str(x)` convertit l'entier ou flottant `x` en une chaîne de caractères

```
>>> str(42)
'42'
>>> str(4.959e+7)
'49590000.0'
```

Conversions entiers - caractères

La fonction `chr(n)` renvoie le caractère dont le code Unicode est `n`

La fonction `ord(s)` affiche le code Unicode du premier caractère de la chaîne `s`

```
>>> chr(65)
'A'
>>> chr(945)
'a'
```

Classe str

Certaines opérations sur les chaînes sont en fait des méthodes de la classe `str`

`s.upper()` : renvoie une copie de la chaîne `s` en majuscules

`s.lower()` : renvoie une copie de la chaîne `s` en minuscules

`s.split(c)` : découpe la chaîne `s` en utilisant le premier caractère de `c` comme séparateur et renvoie les éléments dans un tableau

```
>>> t='Bonjour, ca va ?'
>>> t.upper()
'BONJOUR, CA VA ?'
>>> t.lower()
'bonjour, ca va ?'
>>> t.split(' ')
['Bonjour,', 'ca', 'va', '?']
>>> u=['Oui', 'ca', 'va' ]
>>> "_".join(u)
'Oui_ca_va'
```

Entrées/Sorties élémentaires : la fonction `print`

Une appel `print(e1,...,en)` affiche tous ses arguments anonymes les uns à la suite des autres, séparés par des espaces et avec un retour à la ligne finale

Certains arguments nommés permettent de modifier le comportement de la fonction

- `sep=s` où `s` est la chaîne de séparation (par défaut `' '`)
- `end=s` où `s` est la chaîne de fin de ligne (par défaut `'\n'`)
- `file=d` où `d` est le fichier de sortie (par défaut `sys.stdout`, mais on peut le remplacer par `sys.stderr` ou un fichier)
- `flush=b` où `b` est un booléan qui permet de forcer les écritures dans le fichier (par défaut vaut `False`, les écritures sont faites selon des conditions sytème)

Entrées/Sorties élémentaires : la fonction `input`

Un appel `input(s)` interrompt le programme, affiche la chaîne `s` (le prompt) et attend que l'utilisateur tape une phrase qui doit se terminer par un retour chariot

Un appel `input()` est possible (aucun prompt n'est affiché dans ce cas)

La valeur renvoyée par cet appel sera toujours une chaîne de caractères

```
>>> x = input()
123
>>> type(x)
<class 'str'>
```

La valeur None

Il existe une valeur particulière en Python pour représenter l'absence de valeur : il s'agit de `None`, dont le type est `NoneType`

`None` est la valeur renvoyée par une fonction qui ne fait pas de `return`

Mais `None` est une valeur comme les autres : elle peut être passée en argument d'une fonction, renvoyée en résultat, stockée dans une structure de données, etc.

Remarque : il faut bien distinguer l'initialisation d'une variable à None et la non initialisation d'une variable

```
>>> x = None
>>> print(x)
None
>>> print(y)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'y' is not defined
```

Instructions conditionnelles `if/elif/else`

On présente cette construction à l'aide du programme suivant :

- Compter les points au jeu de Mölkky

Instructions conditionnelles : le jeu de Mölkky

Au jeu de Mölkky, chaque joueur marque à son tour de jeu entre 0 et 12 points, qui viennent s'ajouter à son score précédent

Le premier à atteindre un score de 51 gagne

Mais gare ! Quiconque dépasse le score cible de 51 revient immédiatement à 25 points

On souhaite écrire un programme demandant un score et un nombre de points marqués, et qui affiche le nouveau score ou signale une éventuelle victoire.

Instructions conditionnelles : molkky.py

```
score = int(input("Entrer le score : "))
gain = int(input("Entrer le gain : "))
nouveau_score = score + gain
```

```
if score == 51:
    print("Victoire")
elif score < 51:
    print("Nouveau score :", score)
else:
    print("Nouveau score : 25")
```

Instructions conditionnelles : molkky.py

```
score = int(input("Entrer le score : "))
gain = int(input("Entrer le gain : "))
nouveau_score = score + gain
```

```
if score == 51:
    print("Victoire")
else
    if score < 51:
        print("Tout va bien")
    else:
        print("Dommage")
        score = 25
    print("Nouveau score :", score)
```

Instruction `if/else`

La syntaxe d'une instruction `if/else` est :

```
if expr:
    instruction i1
    instruction i2
    ...
else:
    instruction j1
    instruction j2
    ...
```

`expr` est une expression booléene

Les instructions $\texttt{i}_n$ sont exécutées si `expr` vaut `True`

Les instructions $\texttt{j}_n$ sont exécutées si `expr` vaut `False`

La partie `else:` est optionnelle

Les instructions dans les deux branches doivent être décalées du même nombre d'espaces

Bloc d'instructions

Python est un langage dans lequel l'indentation est significative. On ne peut pas mettre des retours à la ligne ou des espaces n'importe où.

L'indentation indique des blocs d'instructions qui appartiennent au même contexte. Par exemple, le code ci-dessous n'affiche rien

```
x = 45
if x < 10:
    print ("on teste x")
    print ("x est plus petit que 10")
```

Tandis que celui-ci affiche « x est plus petit que 10 »

```
x = 45
if x < 10:
    print ("on teste x")
print ("x est plus petit que 10")
```

L'absence d'indentation mets le deuxième `print` en dehors du `if`

Instruction `if/elif/else`

Il est possible d'avoir plusieurs branches dans une conditionnelle. Chanque branche est introduite par le mot-clé `elif`

```
if expr1:
    bloc b1
elif expr2:
    bloc b2
elif exp3:
    bloc b3
...
else:
    bloc bn
```

Le bloc `bn` de la dernière branche est sélectionné lorsque toutes les expressions $\texttt{expr}_1,\ldots,\texttt{expr}_{n-1}$ sont fausses

Commentaires

Un commentaire en Python commence avec le caractère # et s'étend jusqu'à la fin de la ligne

```
    if x > 0 and x <= 10:
        # x est dans l'intervalle ]0;10]
        y = 1000 / x
```

Fin d'un programme

Un programme Python se termine simplement quand la dernière instruction se termine

Il est cependant possible d'interrompre l'exécution d'un programme à l'aide de la fonction `exit`

Un appel `exit(msg)` arrête définitivement le programme qui l'appelle après avoir affiché le message `msg`

L'instruction assert

Une autre (et meilleure) solution pour stopper un programme consiste à utiliser l'instruction assert

```
assert cond, msg
```

Cette instruction permet de combiner le test d'une condition `cond` et l'interruption du programme avec un message `msg` dans le cas où cette condition n'est pas vérifiée

Si la condition est fausse, le programme lève l'exception `AssertionError`

La boucle `for`

On présente cette construction à l'aide de deux programmes :

- La spirale
- Calcul de la moyenne

Boucle `for` : la spirale

On souhaite écrire un programme dessinant une spirale ayant l'allure suivante, avec un nombre de tours défini par l'utilisateur.

Pour être facilement dessinée et néanmoins harmonieuse, cette spirale est constituée de demi-cercles dont les dimensions augmentent régulièrement : chaque demi-cercle a une épaisseur de 1 supérieure à l'épaisseur du précédent et un rayon qui est le carré de cette valeur.

La boucle for : spirale.py

```
from turtle import *

n = int(input('Nombre de cercle?'))
for i in range(2, 2 * (n+1)):
    width(i)
    circle(i * i, 180)

input('')
```

Boucle `for` : calcul de la moyenne

Nous souhaitons maintenant écrire un programme calculant la moyenne d'un nombre arbitraire de notes.

Ce programme demande d'abord à l'utilisateur le nombre `n` de notes qui vont être saisies, une boucle est alors exécutée `n` fois pour demander à l'utilisateur de saisir chacune des `n` notes et la moyenne est calculée à la fin de la boucle.

Boucle for : moyenne.py

```
n = int(input("Combien d'etudiants ?"))

s = 0

for i in range(n):
    note = int(input('Entrez une note: '))
    s += note

m = s / n

print('La moyenne des',n,'etudiants est de',m)
```

Boucle `for` : généralités

Le langage Python dispose d'une seule boucle `for` qui permet d'itérer sur les éléments d'une collection.

La forme générale de cette boucle de type *for each* est la suivante :

```
for i in col :
    instruction 1
    instruction 2
    ...
```

La variable `i` prend tour à tour les valeurs de la collection `col`.

Le corps de la boucle est matérialisé par l'indentation des instructions qui le compose. Ces instructions peuvent utiliser la valeur du compteur de boucle.

Boucle `for` : spécificités Python

On peut omettre de nommer le compteur de boucle.

```
for _ in range(3):
    print('#')
```

La variable de boucle est accessible après la boucle. Par exemple, le code suivant affichera 3.

```
for i in range(4):
    pass
print('Valeur de i apres la boucle:',i)
```

Le compteur de boucle est modifiable à l'intérieur de la boucle, mais il reprend sa "vraie" valeur à chaque tour de boucle :

```
for i in range(4):
    i = i * 100
    print('i dans la boucle : ',i)
```

Collection d'entiers : la fonction `range`

On implémente une boucle `for` *standard* sur les entiers à l'aide de la fonction `range(i,e,k)` qui renvoie une collection d'entiers compris entre `i` (inclus, 0 par défaut) et `e` (exclu) par pas de `k` (1 par défaut).

```
range(4) --> 0, 1, 2, 3
range(2,5) --> 2, 3, 4
range(2,8,2) --> 2, 4, 6
range(4,0,-1) --> 4, 3, 2, 1
range(6,1,-2) --> 6, 4, 2
```

L'instruction `continue`

À l'intérieur d'une boucle, l'instruction `continue` permet d'indiquer que l'on ne souhaite pas exécuter la fin du tour de boucle en cours, pour passer directement au tour suivant

Les utilisations de `continue` sont assez rares, mais elles sont pertinentes lorsque le traitement normal d'une boucle ne s'applique pas à certains cas d'exception qui doivent simplement être ignorés

```
for i in range(n):
    ...
    if c: continue
    bloc b
```

où `b` dénote un bloc de code quelconque

On obtient une boucle qui à chaque tour exécute le bloc de code `b`, sauf dans les cas particuliers dans lesquels la condition `c` est vérifiée

L'instruction break

On peut stopper l'exécution d'une boucle à l'aide de l'instruction `break`

```
for i in range(n):
    ...
    if c: break
    ...
```

La boucle `for` est arrêtée dès que l'instruction `break` est exécutée, le tour en cours n'étant même pas terminé

Remarque : l'instruction `break` ne peut être utilisée que dans le corps d'une boucle (`for` ou `while`)

La boucle `while`

On présente cette construction à l'aide du programme suivant :

- Racine carrée : La méthode de Héron

Boucle `while` : La méthode de Héron

Pour déterminer la racine carrée du nombre positif a, on calcule la suite x_n définie de la manière suivante :

$$x_{n+1} = \frac{x_n + a/x_n}{2}$$

avec un terme $x_0 > 0$

La suite converge vers $\sqrt{a}$. Pour s'arrêter, il suffit de choisir une valeur de précision ϵ et de stopper le calcul quand $|x_n^2 - a| \leq \epsilon$

Boucle while : heron.py

```
e = 0.01
a = 2
x = 1
while abs(x * x - a) > e :
    x = (x + a / x) / 2
print(x)
```

Boucle `while` : Généralités

Le langage Python dispose d'une boucle `while` similaire aux autres langages de programmation.

La forme générale de cette boucle est la suivante :

```
while cond :
    instruction 1
    instruction 2
    ...
```

Le corps de la boucle est matérialisé par l'indentation des instructions qui le compose.

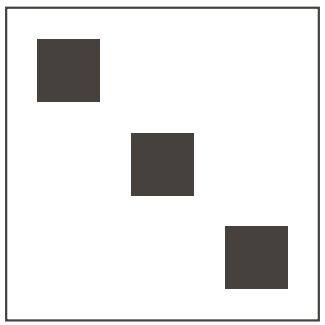
Les fonctions

On présente la définition de fonction à l'aide du programme suivant :

- Dessiner un dé

Les fonctions : dessiner un dé

On souhaite dessiner à l'aide de la bibliothèque Turtle les faces d'un dé, par exemple la face 3



Afin de ne pas écrire plusieurs fois le code pour dessiner des carrés (remplis ou non), on va définir des fonctions.

Les fonctions : de.py

```
from turtle import *

def carre(x, y, n):
    up()
    goto(x, y)
    down()
    for _ in range(4):
        left(90)
        forward(n)

def carre_rempli(x, y, n):
    begin_fill()
    carre(x, y, n)
    end_fill()

carre(50, -50, 100)
carre_rempli(10, -10, 20)
carre_rempli(40, -40, 20)
carre_rempli(-20, 20, 20)
input('')
```

Appels de fonctions

En Python, un appel de fonction ce note $\texttt{f}(\texttt{e}_1, \ldots, \texttt{e}_n)$

Les arguments d'une fonction peuvent être donnés, dans n'importe quel ordre, en utilisant les arguments nommés. On peut dans ce cas appeler une fonction avec $\texttt{f}(\texttt{x}_{k_1}\texttt{=e}_{k_1}, \ldots, \texttt{x}_{k_n}\texttt{=e}_{k_n})$, où les arguments $\texttt{x}_{k_i}$ sont tous différents.

Lors d'un appel, on peut aussi mélanger arguments nommés et arguments anonymes, mais il faut toujours donner les arguments anonymes en premier, c'est-à-dire des appels de la forme $\texttt{f}(\texttt{e}_1, \ldots, \texttt{e}_p, \texttt{x}_{k_1}\texttt{=e}_{k_1}, \ldots, \texttt{x}_{k_{(n-p)}}\texttt{=e}_{k_{(n-p)}})$

Définition d'une fonction

En Python, le mot-clé `def` premet de définir une fonction

```
def f(x1, x2, ..., xk) :
    instruction i1
    instruction i2
    instruction i3
    ...
```

L'instruction `return` permet de quitter une fonction en renvoyant la valeur `None`

L'instruction `return e` quitte la fonction en renvoyant la valeur de l'expression `e`

Passage par valeur

Python fait du passage par valeur des arguments aux fonctions.

Cela signifie que les arguments sont copiés sur la pile.

Si une fonction modifie ses arguments, les modifications sont locales à la fonction.

```
def f(a):
    a = 42
    print(a)

a = 18
f(a)      #affiche 42
print(a) #affiche 18
```

Dans une fonction, les paramètres se comportent comme des variables locales.

Variables locales

```
def sumproduct(a, b, c):
    tmp = a + b
    tmp2 = tmp * c
    return tmp2
```

Les variables `tmp` et `tmp2` sont des variables locales à la fonction

- On ne peut pas y accéder depuis l'extérieur
- Elles *commencent à exister* quand on rentre dans la fonction
- Elles *cessent d'exister* quand on sort de la fonction

Variables globales

Une variable définie en dehors d'une fonction est une variable globale

```
N = 1
def suivant(x):
    return x + N

print(suivant(1))   # affiche 2
N = 17
print(suivant(1))   # affiche 18
```

Attention :

```
N = 42
def change():
    N = 666
    print ("In", N)
    return

change()            # affiche 'In 666'
print ("Out", N)    # affiche 'Out 42'
```

Variables globales utilisées dans une fonction

Lorsque l'on écrit `x = e` dans une fonction (i.e. si `x = e` apparaît n'importe où dans la fonction)

- Si l'instruction `global x` a été donnée au début de la fonction alors la variable globale `x` sera crée ou modifiée
- Sinon la variable locale `x` sera crée ou modifiée

Lorsqu'on utilise une variable `x` dans une fonction

- Si une variable locale `x` existe, sa valeur est utilisée
- Sinon si une variable globale `x` existe (et que `x=e` n'apparaît pas dans la fonction), sa valeur est utilisée
- Sinon erreur : variable non définie

Variables globales : exemples

```
X = 1
Y = 2
def f1(a, b):
    X = a # X est locale
    Y = b # Y est locale

def f2(a, b):
    global X, Y
    X = a # X globale et modifiee
    Y = b # Y globale et modifiee

def f3():
    return X+Y # pas de variable locale,
               # va chercher les variables globales

def f4(a):
    X = X + a  # X = ... indique que X est locale,
               # mais dans X + a, X n'est pas
               # encore definie, donc erreur !
    return X
```

Arguments optionnels

On peut ajouter des arguments optionnels à une fonction en les déclarant avec une valeur par défaut :

```
def f(x, y=42, z=10):
    return x+y+z
```

On peut alors utiliser la fonction en lui donnant ou non des valeurs pour ces arguments :

```
f(1) # renvoie 53
f(1,z=2) # renvoie 45
f(1,y=3) # renvoie 14
f(1,2,3) # renvoie 6
```

Attention : les arguments optionnels doivent être définis en dernier.

Les conteneurs

En Python, on appelle conteneur un objet qui sert à contenir d'autres objets

Les chaînes de caractères sont des conteneurs qui stockent des caractères

Python propose d'autres conteneurs, plus génériques, dont les 4 principaux sont :

- Les tuples (ou n-uplets)
- Les tableaux (ou listes)
- Les dictionnaires
- Les ensembles

Les tableaux (type `list`)

Un tableau est une collection ordonnée, finie et hétérogène de valeurs avec un accès efficace à n'importe quel élément

- `[e1, ..., en]` définition d'un tableau
- `t[i]` accès au ième élément du tableau `t` (indice de départ 0) ; on accède aux éléments à rebours avec des indices négatifs.
- `t[i] = e` mise à jour du ième élément du tableau `t`
- `len(t)` longueur du tableau `t`

```
>>> tab = [1, 3, 5, 4, 19, 2]
>>> tab
[1, 3, 5, 4, 19, 2]
>>> tab[4] + tab[-3]
23
>>> tab[4] = 42
>>> tab
[1, 3, 5, 4, 42, 2]
>>> t1 = [1, 'toto', 4.5]
>>> t2 = [t1, [], 4]
```

Les tableaux : opérations avancées

- `t1 + t2` concaténation de deux tableaux ; renvoie un nouveau tableau avec les éléments de `t1` et `t2` bout à bout
- `t * n` concatène `n` fois le tableau `t` avec lui même
- `t[i:j]` renvoie une copie du tableau prise entre les indices `i` (inclus) et `j` (exclu)

```
>>> t1 = [1, 2, 3]
>>> t2 = [4, 5, 6]
>>> t1 + t2
[1, 2, 3, 4, 5, 6]
>>> t3 = t1 + t2
>>> t3[0] = 10
>>> t3
[10, 2, 3, 4, 5, 6]
>>> [0] * 10
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
>>> t3[2:4]
[3, 4]
```

Les tableaux : structure redimentionnable

Les tableaux Python sont des tableaux redimentionnables.

Ils possèdent une interface objet impérative.

Parmis les méthodes on trouve :

- `t.append(e)` ajoute l'élément `e` en fin de tableau
- `t.pop()` retire et renvoie le dernier élément du tableau
- `t.insert(i, e)` ajoute l'élément `e` à l'indice `i` dans le tableau `t`
- `t.clear()` vide le tableau `t`
- `t.remove(e)` retire la première occurrence de `e` dans `t`

Les tableaux : Complexités des opérations

Opération	Exemple	Classe
Load	`l[i]`	O(1)
Store	`l[i] = v`	O(1)
Length	`len(l)`	O(1)
Append	`l.append(5)`	O(1)
Pop	`l.pop()`	O(1)
Slice	`l[a:b]`	O(b-a)
Delete	`del l[i]`	O(n)
Copy	`l.copy()`	O(n)
Remove	`l.remove(v)`	O(n)
Popi	`l.pop(i)`	O(n)
Multiply	`k*l`	O(k $\times$ n)

Tableaux et fonctions

Comme nous l'avons vu, Python fait du passage par valeur, mais la valeur d'un tableau est son adresse en mémoire

```
def f(tab):
    tab[0] = 42
tab = [1,2,3]
f(tab)
print(tab) #affiche [42, 2, 3]
```

On peut donc modifier les cases du tableau, mais pas la variable contenant le tableau :

```
def f(tab):
    tab = "toto"
tab = [1,2,3]
f(tab)
print(tab) #affiche [1, 2, 3]
```

Les tuples (1)

Python propose le type de donnée de tuple (ou n-uplet)

On écrit les expressions en utilisant des parenthèses et des virgules

```
>>> point = (1.5, -3.19)
>>> point
(1.5, -3.19)
>>> point[0]
1.5
>>> point[1]
-3.19
>>> x, y = point
>>> x + y
-1.69
```

Attention : Les tuples ne sont pas modifiables

```
>>> point[0] = 2.2
 Traceback (most recent call last):
 File "", line 1, in <module>
 TypeError: 'tuple' object does not support item
                                assignment
```

Les tuples (2)

- La fonction `len(t)` renvoie le nombre de composantes
- `()` est le tuple de taille 0
- `(v, )` est un tuple de taille 1 contenant v (La virgule seule est obligatoire, sinon `(v)` est compris comme `v` entouré de parenthèses « mathématiques »
- On peut utiliser le + et le * comme pour des tableaux

```
>>> t = (1, 2) + (3, 4, 5)
>>> len(t)
5
>>> t
(1, 2, 3, 4, 5)
>>> (42, ) * 5
 (42, 42, 42, 42, 42)
>>> (42) * 5
210
>>> (x,y) = (4, ('titi', 4.5, False))
>>> y[1]
'titi'
```

Les dictionnaires (1)

Python propose le type de donnée de dictionnaire

Il est similaire aux tableaux, mais les indices peuvent être n'importe quel type de données Python (non modifiable)

On définit un dictionnaire vide par { }

On peut construire un dictionnaire avec la notation

{ k1:v1, ..., kv:vn }

```
>>> jours = { 'lundi':1, 'mardi':2, 'mercredi':3 }
>>> jours['mardi']
2
>>> jours
{'lundi':1, 'mardi':2, 'mercredi':3}
>>> jours['jeudi'] = 4
>>> jours
 {'lundi':1, 'mardi':2, 'mercredi':3, 'jeudi' : 4}
>>> jours['jeudi'] = 42
>>> jours
 {'lundi':1, 'mardi':2, 'mercredi':3, 'jeudi' : 42}
```

Les dictionnaires (2)

Accéder à une clé inexistante est similaire à faire un accès invalide dans un tableau

L'opérateur `in` permet de tester si une clé est dans le dictionnaire

```
>>> jours['toto']
Traceback (most recent call last):
  File "", line 1, in <module>
KeyError: 'toto'
>>> 'mardi' in jours
True
>>> 'toto' in jours
>>> jours
False
```

On peut utiliser d'autres types de valeur pour les clés (entiers, booléens). L'utilisation la plus fréquente reste les chaînes de caractères.

Attention, comme les tableaux, les dictionnaires sont mutables !

Les dictionnaires : des structures itérables

On peut facilement itérer sur les éléments (clés ou valeurs) d'un dictionnaire à l'aide d'une boucle `for`

```
>>> for k in jours:
        print(k, jours[k])
lundi 1
mardi 2
mercredi 3

>>> for k, j in jours.items():
        print(k, jours[k])
lundi 1
mardi 2
mercredi 3
```

Les ensembles (1)

Python fournit également la structure d'ensemble : structure d'éléments non ordonnés, sans duplications et hétérogène

```
>>> s1 = {1, 4, 1, 2, 10}
>>> s1
{1, 2, 10, 4}
>>> s2 = { 'toto', 10, 4.5 }
```

Attention : Les éléments d'un ensemble doivent être non modifiables

```
>>> { [1], [2] }
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

Cardinalité $|s|$: `len(s)`

Test d'appartenance $v \in s$: `v in s`

Les ensembles (2)

Attention : Certaines opérations sur les ensembles sont impératives et d'autres persistantes

Exemples d'opérations avec modifications en place :

```
>>> s1.add('5')
>>> s1
{1, 2, 4, 5, 10}
>>> s1.remove(2)
>>> s1
{1, 4, 5, 10}
```

Exemples d'opérations qui créent de nouvelles structures

```
>>> s1 = {1,4,5,10}
>>> s2 = s1.union({2,3})
>>> s3 = s1.difference({1,10})
>>> s2.intersection(s3)
{4, 5}
```

La bibliothèque Turtle (1)

Les instructions de la bibliothèque Turtle font se déplacer une tortue munie d'un crayon à la surface d'une feuille virtuelle

La tortue commence au point $(0, 0)$ situé au centre de l'écran. Elle est orientée vers la droite (axe des abscisses). Les coordonnées et distances sont mesurées en pixels et les angles en degrés. Les arcs de cercles sont parcourus dans le sens trigonométrique (si le rayon est positif, sens horaire sinon)

Instruction	Description
`goto(`x`, `y`)`	aller au point de coordonnées (x, y)
`forward(`d`)`	avancer de la distance d
`backward(`d`)`	reculer de la distance d
`left(`a`)`	pivoter à gauche de l'angle a
`right(`a`)`	pivoter à droite de l'angle a
`circle(`r`, `a`)`	tracer un arc de cercle d'angle a et de rayon r
`dot(`r`)`	tracer un point de rayon r

La bibliothèque Turtle (2)

Pour modifier les dessins produits par chacun des déplacements

Instruction	Description
`up()`	relever le crayon (et interrompre le dessin)
`down()`	redescendre le crayon (et reprendre le dessin)
`width(`e`)`	fixer à e l'épaisseur du trait
`color(`c`)`	sélectionner la couleur c pour les traits
`begin_fill()`	activer le mode remplissage
`end_fill()`	désactiver le mode remplissage
`fillcolor(`c`)`	sélectionner la couleur c pour le remplissage

Les tracés sont faits en noir (`black`, par défaut) avec une épaisseur d'un pixel (autres couleurs : `'blue'`, `'green'`, `'yellow'`, ..., ou `color(R, V, B)`)

Toute l'aire contenue à l'intérieur de la trajectoire de la tortue pendant une période de temps où le mode remplissage est activé prend la couleur choisie pour le remplissage

EXCEPTIONS, ENTRÉES-SORTIES, MODULES

Les exceptions

Python dispose d'un mécanisme de gestion des erreurs basé sur les exceptions

Lorsqu'un programme exécute une instruction qui génère une erreur, celle-ci est transformée en une *exception* qui interrompt le programme en cours

Par exemple, une exception est levée quand la fonction `int` est appelée avec une chaîne de caractères qui ne représente pas un entier

```
>>> int('abc')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '
                                    abc'
```

Lever une exception

L'utilisateur a la possibilité de lever lui-même une exception à l'aide de l'instruction `raise`

```
>>> raise NameError('message')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: message
```

Rattrapage d'exceptions (1)

On veut parfois vouloir gérer une erreur au moment où elle se produit. On peut pour cela utiliser la construction `try /except`, appelée *handler* d'exceptions

```
try:
  bloc i
except E:
  bloc e
```

Sémantique : Le bloc `i` d'instructions est exécuté. Si une instruction de ce bloc lève l'exception alors il s'interrompt. Si cette exception est `E` alors le bloc `e` d'instructions est exécuté. Si le bloc `i` lève une autre exception que `E`, alors cette exception n'est pas capturée et elle remonte au prochain *handler*. S'il n'y a aucun *handler* pour capturer cette exception, le programme se termine.

Rattrapage d'exceptions (2)

On peut aussi omettre d'indiquer l'exception à rattraper

Dans ce cas, le *handler* d'exception rattrapera toutes les exceptions levées dans le bloc

```
try:
  bloc i
except:
  bloc e
```

Il est également possible de rattraper plusieurs exceptions différentes et de leur associer des blocs d'instructions différents

```
try:
  bloc i
except E1:
  bloc e1
except E2:
  bloc e2
...
```

Rattrapage d'exceptions (3)

Il est également possible de manipuler une exception en la nommant explicitement

```
try:
  bloc i
except E as ex:
  bloc e
```

Cela permet de stocker l'exception, d'extraire ses arguments, de l'imprimer, etc.

```
>>> try:
      raise NameError('msg')
    except NameError as e:
      exn = e
>>> exn
NameError('msg')
```

Rattrapage d'exceptions (4)

Dans une expression `except as e`, la variable e représente aussi directement les arguments de l'exception

```
>>> try:
      raise NameError('un message')
    except NameError as e:
      print(e)
un message
```

D'une manière générale, la variable e représente le n-uplet des arguments de l'exception

Définir des exceptions (1)

En Python, on peut définir des exceptions en définissant une classe qui hérite de la classe `Exception` (plus de détails sur les classes et l'héritage dans le cours dédié aux objets)

```
class MyError(Exception):
    pass

def f(x):
    if x == 0:
        raise MyError
    return 100/x
```

Définir des exceptions (2)

Les exceptions définies par l'utilisateur peuvent aussi avoir des arguments.

```
class MyError(Exception):
    def __init__(self,x,y):
        self.X = x
        self.Y = y

def f(x):
    if x == 0:
        raise MyError(x,'erreur grave')
    return 100/x

try:
   f(0)
except MyError as e:
    exn = e
```

A la fin, la variable `exn` vaut `MyError(0, 'erreur grave')` et on peut accéder aux arguments de l'exception avec `exn.X` et `exn.Y`

Lecture et écriture de fichiers

En python, un appel `open(chemin, mode)` permet d'ouvrir un fichier pour le lire

- `chemin` est une chaîne de caractères contenant le chemin vers le fichier
- Le chemin peut être absolu (commencer par `/`) ou relatif
- Il ne peut pas contenir de caractères spéciaux du shell tel que `~` ou des motifs *glob* (`[a-z]*.txt`)
- `mode` est une chaîne de caractères indiquant le mode d'ouverture :
  - `r` Le fichier est ouvert en lecture seule.
  - `w` Le fichier est ouvert en écriture seule. Le fichier est créé s'il n'existe pas et vidé de son contenu s'il existe.
  - `w+` Comme w, mais aussi accès en lecture.
  - `a` Le fichier est ouvert en écriture et lecture. qLe fichier est créé s'il n'existe pas. Le contenu est conservé si le fichier existe.
  - `a+` Comme a, mais aussi accès en lecture.

Fichiers : gestion des erreurs

Les opérations sur les fichiers peuvent provoquer des erreurs.
Celles-ci sont généralement exprimées en levant des exceptions

- Si le chemin n'existe pas
- Si le nom indiqué pointe sur un répertoire (et pas un fichier)
- Si on a pas les droits nécessaires (par exemple pas les droits en écriture et qu'on ouvre avec le mode `w`)
- Si on essaye d'écrire dans le fichier et qu'il n'y a plus de places sur le disque

On pourra utiliser `try /except` pour rattraper certaines de ces erreurs

Opérations sur les fichiers (1)

Le résultat de open(...) est une valeur spéciale, appelée descripteur de fichier. C'est un *objet opaque* qui possède de nombreuses opérations. On se limite aux plus simples

On suppose que dans le répertoire courant, on a un fichier `test.txt`

```
>>> f = open("test.txt", "r")
>>> f
<_io.TextIOWrapper name='test.txt' mode='r' encoding='
                                    UTF-8'>
>>> lines = f.readlines()
>>> lines
['Ceci est un fichier\n', 'qui contient plusieurs\n',
                                    'lignes\n']
>>>
```

L'opération `f.readlines()` renvoie le tableau de toutes les lignes du fichier f. Les retours à la ligne sont conservés.

Opérations sur les fichiers (2)

```
>>> for i in range(len(lignes)):
        lignes[i] = lignes[i].upper()
>>> f2 = open('fichier2.txt','w')
>>> f2.writelines(lines)
>>> f2.close()
```

Après l'exécution de ce programme, le fichier `fichier2.txt` contient le même contenu que `fichier.txt`, mais avec toutes les lettres en majuscules

L'opération `f2.writelines(lines)` écrit le tableau de chaînes de caractères `lines` dans le fichier `f2`. Lorsqu'on a fini, il faut refermer le fichier en appeleant `f2.close()` sinon il se peut que certaines lignes ne soient pas écrites dans le fichier

Les gestionnaires de contextes

Lors de l'utilisation de certaines ressources (fichiers, connexions réseau, ...) il est souvent utile de pouvoir dire « lorsque la ressource n'est plus utilisée, libérer la ressource »

Une construction spéciale existe en Python pour cela :

```
with open("fichier2.txt", "rw") as f2:
    lines = f2.readlines()
    for l in range(len(lines)):
        lines[i] = lines[i].upper()
    f2.writelines(lines)
```

Dans le code ci-dessus, le fichier `f2` est refermé (avec `f2.close()`) lorsque l'on quitte le bloc with, de quelque façon que ce soit (fin du bloc, exception, return, ...)

Les modules

Un module Python est simplement un fichier `.py` qui contient des définitions (de fonctions ou de variables) et des instructions.

Par défaut, on ne peut référencer que des fonctions et variables du fichier (i.e du module) dans lequel on se trouve

La directive `import` permet d'importer tout ou partie des fonctions et variables d'un module

La commande `import`

La *directive* suivante ajoute le nom du module `module1` dans l'environnement des variables globales

```
import module1
```

Il est ensuite possible d'utiliser les fonctions ou valeurs définies dans ce fichier. Par exemple, si le fichier `module1.py` est le suivant :

```
x = 42
def f(x):
    return x+1
```

On peut utiliser la variable `x` et le fonction `f` définies dans ce module en les préfixant par `module1.` de la manière suivante :

```
import module1
y = module1.x + module1.f(100)
print(y)
```

Effets de la commande `import`

Lorsqu'on écrit `import module1`, l'interprète Python cherche dans le répertoire courant, puis dans les répertoire systèmes (dans cet ordre par défaut) un fichier `module1.py`

Les instructions du module importé sont exécutées au moment de l'import

Par exemple, dans l'exemple donné dans le slide précédent, si `module1.py` contient :

```
x = 42
print(x)
def f(x):
    return x+1
```

alors la valeur `42` sera affichée, puis `y` sera calculée et enfin l'instruction `print(y)` sera exécutée

Import partiel

Il est possible d'importer une sélection de valeurs (fonctions) dans un module. Pour cela, on utilisera la directive `from ... import ...`

```
from math import sin, sqrt, pi
y = sin(pi / 2)
z = sqrt(499)
```

Dans le code ci-dessus, seules `sin`, `sqrt` et `pi` sont visibles.

La forme `from foo import *` importe tous les symboles, sans préfixe

On peut également donner un autre nom au module importé à l'aide de la directive `import <module> as <nom>`

Portée des noms avec un import

Les noms des valeurs importées par un module cachent les valeurs de même nom importées précédemment

Exemple avec 3 modules

module2.py

```
def f1(x,y):
    return (x+y)
def f2(x):
    return (x * 2)
```

module3.py

```
def f1(x): return (100)
```

module4.py

```
def f2(x): return (x + 100)
```

```
from module2 import f1, f2
import module3 as m # ne cache pas f1, car on doit
                                appeler m.f1
from module4 import f2 # cache la fonction f2 de
                                module2
```

Pourquoi utiliser des modules ?

Il y a deux aspects *contradictoires* :

- Toutes les fonctions doivent avoir un nom distinct
- On doit utiliser des noms les plus courts mais les plus descriptifs possibles. *Your variable names should be short, and sweet and to the point* (L. Torvalds)
  Exemple :
  - Le module `math` de Python définit une fonction logarithme. Elle s'appelle `log`
  - Le module `logging` de Python définit une fonction permettant d'afficher des messages d'erreurs dans la console et dans des fichiers. Elle s'appelle `log` (c'est le terme en anglais)

Sans système de module, on aurait du utiliser une convention arbitraire par exemple `math_log` et `console_log`. C'est moche et source de bugs.

Pourquoi `import *` c'est mal ?

```
from logging import log, WARNING, ERROR
from math import *
...
log (WARNING, "attention !") #Erreur utilise math.log
...
```

Dans le code ci-dessus, on a masqué involontairement la fonction `log` du module `logging`, par une fonction qui fait complètement autre chose

Le module sys

Le module `sys` fournit un accès aux variables propres à l'interpréteur Python

`https://docs.python.org/fr/3/library/sys.html`

Ce module permet de récupérer les arguments passés à un programme Python depuis la ligne de commande d'un terminal

test.py

```
import sys
print(sys.argv)
```

```
> python3.9 test.py 42 toto 10.5
['test.py', '42', 'toto', '10.5']
```

Ce module définit également les fichiers spéciaux `stdout`, `stin` et `stderr`, pour représenter la sortie standard, l'entrée standard et la sortie d'erreurs

Le module `math`

Le module `math` fournit un ensemble de fonctions mathématiques

`https://docs.python.org/fr/3/library/math.html`

On y trouve des fonctions

- arithmétiques (`ceil`, `floor`, `gcd`, ...)
- logarithme et exponentielle (`log`, `exp`, `sqrt`,...)
- trigonométiques (`cos`, `sin`, `tan`,...), des constantes (`pi`, `e`, ...)

Le module re

Le module re permet de manipuler des expressions régulières

https://docs.python.org/fr/3/library/re.html

findall(r,s) permet de trouver tous les mots de s qui correspondent à l'expression régulière r

split(r,s) découpe la chaîne s selon l'expression régulière r

Exemple :

```
>>> import re
>>> re.findall('b[^ ,]+', 'bob, comment va brice ?')
['bob', 'brice']
>>> re.split('[ ,-]+', 'Comment allez-vous?')
['Comment', 'allez', 'vous?']
```

Compléments (1/2)

Python propose une construction `if/then/else` dans la catégorie syntaxique des expressions

```
>>> x = e1 if c else e2
```

La valeur de l'expression `e1 if c else e2` est `e1` si l'expression `c` vaut `True` et `e2` sinon

Compléments (2/2)

Python permet de définir de tableaux en compréhension

```
>>> l1 = [1, 2, 3, 4, 5, 6, 7, 8 ,9]
>>> l2 = [ x * 2 for x in l1 ]
>>> l3 = [ x * 2 for x in range(1,10) ]
>>> l4 = [ (i,j) for i in range(0,4) \
                 for j in range(0,3) ]
>>> l5 = [ (i,j) for i in range(0,6) \
                 for j in range(0,6) if i < j]
```

La syntaxe (simplifiée) est : `[e1 for x in e2 if c]`

- e2 est une collection et `x` une variable qui va prendre tour à tour les valeurs des éléments de e2
- e1 est une expression (qui peut contenir la variable `x`) dont le résultat sera stocké dans la liste
- c est une expression booléenne (qui peut utiliser x) ; si c est fausse, la valeur de `x` n'est pas utilisée et on passe à la suivante

PROGRAMMATION FONCTIONNELLE EN PYTHON

Notions introduites

- Fonctions passées en arguments
- Fonctions renvoyées comme résultats
- Structures de données immuables
- Programmation avec itérateurs

Qu'est-ce que la programmation fonctionnelle ?

On peut tenter de répondre à cette question par une autre question

Qu'est-ce que la programmation fonctionnelle ?

On peut tenter de répondre à cette question par une autre question

Quelles seraient les conséquences sur votre style de programmation si les variables n'étaient plus modifiables ?

Qu'est-ce que la programmation fonctionnelle ?

On peut tenter de répondre à cette question par une autre question

Quelles seraient les conséquences sur votre style de programmation si les variables n'étaient plus modifiables ?

- Les boucles `while` deviennent nécessairement inutiles car on y entre jamais ou en sort jamais !
- Les boucles `for` deviennent aussi *presque* inutiles car on peut tout de même y entrer et en sortir, mais on ne peut pas faire grand chose dans le corps de la boucle.
- Toutes les opérations qui modifiaient des structures de données en place doivent maintenant renvoyer de nouvelles structures

Réaliser des calculs sans boucles

Pour calculer, on va devoir utiliser uniquement des fonctions.

Réaliser des calculs sans boucles

Pour calculer, on va devoir utiliser uniquement des fonctions.

Exemple :

$$\sum_{i=1}^{10} i$$

Réaliser des calculs sans boucles

Pour calculer, on va devoir utiliser uniquement des fonctions.

Exemple :

$$\sum_{i=1}^{10} i$$

```
x = 0
for i in range(1,11):
  x = x + i
```

```
def somme(i):
  if i == 10:
    return i
  else:
    return i + somme(i+1)
```

Des fonctions récursives locales

Pour réaliser des boucles, on utilise des fonctions récursives locales

Des fonctions récursives locales

Pour réaliser des boucles, on utilise des fonctions récursives locales

Par exemple, on peut transformer la fonction suivante :

```
def somme_tab(t):
    v = t[0]
    for i in range(1, len(t)):
        v = v + t[i]
    return v
```

en une fonction contenant un fonction locale :

```
def somme_tab(t):
    def boucle(i,v):
        if i == len(t): return v
        else:
            return boucle(i+1, v+t[i])
    return boucle(1,t[0])
```

La programmation fonctionnelle

C'est donc un styte de programmation où :

- Aucune variable n'est modifiable
- Les structures de données sont persistantes, c'est-à-dire observationnellement immuables
- On utilise uniquement des fonctions pour réaliser des calculs (plus de boucles)

Mais c'est plus que ça :

- Fonctions d'ordre supérieur
- Fonctions anonymes
- Fermetures
- Applications partielles
- ...

FONCTIONS D’ORDRE SUPÉRIEUR

Les fonctions sont des valeurs à part entière

Les fonctions sont des valeurs comme les autres

Une fonction peut être :

- stockée dans une structure de donnée (n-uplets, listes etc.)
- passée en argument à une autre fonction
- retournée comme résultat d'une fonction

Les fonctions prenant des fonctions en arguments ou rendant des fonctions en résultat sont dites **d'ordre supérieur**

Structures de données contenant des fonctions

On peut stocker des fonctions dans n'importe quelle structure de données.

Par exemple, dans un n-uplet :

```
>>> def f(x) : return x+1
>>> def g(x) : return x+x

>>> p = (f,g)
>>> p[0](p[1](4))
17
```

Fonctions anonymes (1/2)

Les fonctions étant des objets comme les autres, on peut les créer sans avoir à leur donner un nom.

Pour cela, on utilise la notation `lambda` pour créer des fonctions anonymes

Fonctions anonymes (1/2)

Les fonctions étant des objets comme les autres, on peut les créer sans avoir à leur donner un nom.

Pour cela, on utilise la notation `lambda` pour créer des fonctions anonymes

```
lambda <arguments> : <expression>
```

Fonctions anonymes (1/2)

Les fonctions étant des objets comme les autres, on peut les créer sans avoir à leur donner un nom.

Pour cela, on utilise la notation `lambda` pour créer des fonctions anonymes

```
lambda <arguments> : <expression>
```

Par exemple, la fonction $x \mapsto x + 1$ est définie de cette manière :

```
lambda x: x+1
```

Remarque : l'expression à droite du ':' ne contient pas de return

Fonctions anonymes (2/2)

On pourra donc écrire de manière équivalente :

```
def f(x) : return x + 1
```

et

```
f = lambda x: x+1
```

Fonctions anonymes (2/2)

On pourra donc écrire de manière équivalente :

```
def f(x) : return x + 1
```

et

```
f = lambda x: x+1
```

Cette notation se généralise aux fonctions à plusieurs arguments. Ainsi, la fonction $x, y \mapsto x + y$ s'écrira

```
lambda x,y: x+y
```

Fonctions anonymes (2/2)

On pourra donc écrire de manière équivalente :

```
def f(x) : return x + 1
```

et

```
f = lambda x: x+1
```

Cette notation se généralise aux fonctions à plusieurs arguments. Ainsi, la fonction $x, y \mapsto x + y$ s'écrira

```
lambda x,y: x+y
```

Et on pourra donc écrire de manière équivalente :

```
def g(x,y) : return x+y
```

ou

```
g = lambda x,y: x+y
```

Fonctions comme arguments

- Certaines fonctions prennent naturellement des fonctions en arguments
- Par exemple, les notations mathématiques telles que la sommation $\Sigma_{i=1}^{n} f(i)$ se traduisent facilement si l'on peut utiliser des arguments fonctionnels

```
>>> def somme(f,n):
        if n <= 0: return 0
    else:
        return f(n) + somme(f, n-1)

>>> somme(lambda x: x * x * x, 5)
225
```

Exemple

On souhaite généraliser la fonction pour calculer la somme des éléments d'un tableau `t` en une fonction qui applique un opérateur quelconque op aux éléments du tableau :

$$\texttt{t[0]}\ op\ \texttt{t[1]}\ op\ \cdots\ op\ \texttt{t}[\texttt{len(t)} - 1]$$

Exemple

On souhaite généraliser la fonction pour calculer la somme des éléments d'un tableau `t` en une fonction qui applique un opérateur quelconque op aux éléments du tableau :

$$\texttt{t[0]}\ op\ \texttt{t[1]}\ op\ \cdots\ op\ \texttt{t}[\texttt{len(t)}-1]$$

```
>>> def calcul(op,t):
        def boucle(i,v):
            if i == len(t):
                return v
            else:
                return boucle(i+1, op(t[i],v))
        return boucle(1,t[0])

>>> calcul(lambda x,y: x * y, [1, 2, 3, 4])
24
```

Fonctions comme résultat

Les fonctions peuvent également être renvoyées comme résultat.

Par exemple, pour approcher la dérivée $f'(x)$ d'une fonction f avec un taux d'accroissement

$$\frac{f(x+h) - f(x)}{h}$$

pour un h suffisamment petit, on pourra écrire la fonction suivante en Python :

```
>>> def derive(f):
        h = 1e-7
        return lambda x: (f(x + h) - f(x)) / h

>>> d = derive(math.sin)
>>> d(0)
0.9999999999999983
```

Fonctions à plusieurs arguments (1/3)

La notion de fonction à plusieurs arguments est *plus subtile* qu'il n'y parait.

Fonctions à plusieurs arguments (1/3)

La notion de fonction à plusieurs arguments est *plus subtile* qu'il n'y parait.

Quelles sont les différences entre les fonctions `f`, `g` et `h` définies de la manière suivante ?

```
>>> f = lambda x,y : x+y

>>> g = lambda p : p[0]+p[1]

>>> h = lambda x: lambda y : x+y
```

Fonctions à plusieurs arguments (2/3)

```
>>> f(4,5)
9
```

Fonctions à plusieurs arguments (2/3)

```
>>> f(4,5)
9
```

```
>>> g(4,5)
Traceback (most recent call last):
File "...", line 1, in <module>
g(4,5)
TypeError: <lambda>() takes 1 positional argument but
                              2 were given
```

Fonctions à plusieurs arguments (2/3)

```
>>> f(4,5)
9
```

```
>>> g(4,5)
Traceback (most recent call last):
File "...", line 1, in <module>
g(4,5)
TypeError: <lambda>() takes 1 positional argument but
                                2 were given
```

La fonction g ne prend donc qu'un unique paramètre, à savoir la paire p.

Pour l'appeler correctement, il faut donc lui donner un seul argument :

```
>>> g( (4,5) )
9
```

Fonctions à plusieurs arguments (3/3)

```
>>> h(4,5)
Traceback (most recent call last):
File "...", line 1, in <module>
h(4,5)
TypeError: <lambda>() takes 1 positional argument but
                                2 were given
```

Là encore, la fonction `h` ne prend donc qu'un unique paramètre

Comment faire pour réaliser le calcul `x + y` ?

APPLICATION PARTIELLE

Des fonctions de fonctions

L'expression suivante

```
lambda x: lambda y : x + y
```

est une fonction qui prend une valeur `x` en argument et qui renvoie comme résultat une fonction qui prend une valeur `y` en argument, cette dernière renvoyant `x+y` comme résultat.

```
>>> plus = lambda x: lambda : x + y
>>> plus(4)(5)
9
```

Des fonctions de fonctions

L'expression suivante

```
lambda x: lambda y : x + y
```

est une fonction qui prend une valeur `x` en argument et qui renvoie comme résultat une fonction qui prend une valeur `y` en argument, cette dernière renvoyant `x+y` comme résultat.

```
>>> plus = lambda x: lambda : x + y
>>> plus(4)(5)
9
```

De manière équivalente, on peut définir `plus` de la manière suivante :

```
def plus(x): return lambda y : x + y
```

Application partielle (1/2)

Puisque la fonction `plus` n'attend qu'un seul argument, que vaut `plus(4)` ?

Application partielle (1/2)

Puisque la fonction `plus` n'attend qu'un seul argument, que vaut `plus(4)` ?

```
>>> h = plus(4)
>>> h(5)
9
>>> h(10)
14
```

Nous venons simplement d'appliquer partiellement la fonction `plus`

Application partielle (2/2)

L'application partielle nous permet par exemple de "spécialiser" des fonctions

```
>>> def derive(h):
        return lambda f: lambda x: (f(x + h) - f(x)) /
                                   h

>>> der1 = derive(1e-10)
>>> der2 = derive(1e-7)
>>> f = der1(math.sin)
>>> g = der2(math.sin)
>>> f(0)
1.0
>>> g(0)
0.9999999999999983
```

La notion de fermeture (1/2)

Considérons le programme suivant :

```
def plus(x) : return lambda y: x+y

>>> h = plus(5)
>>> h(4)
9
```

La notion de fermeture (1/2)

Considérons le programme suivant :

```
def plus(x) : return lambda y: x+y

>>> h = plus(5)
>>> h(4)
9
```

Où est stockée la variable `x` dans la fonction `h` ?

La notion de fermeture (1/2)

Considérons le programme suivant :

```
def plus(x) : return lambda y: x+y

>>> h = plus(5)
>>> h(4)
9
```

Où est stockée la variable `x` dans la fonction `h` ?

Elle devrait avoir disparu de la mémoire d'après le modèle d'exécution avec une pile

La notion de fermeture (2/2)

Une fermeture est une fonction avec un état interne. Cet état est constitué de la valeur des variables libres du corps de la fonction.

Python construit une fermeture en capturant la valeur des variables dont la durée de vie échappe à la portée statique des variables.

Dans l'exemple précédent, `x` est une variable libre de la fonction `lambda y: x+y`

Une fermeture est une sorte de paire, avec une composante qui représente l'adresse mémoire du corps de la fonction, et une autre qui pointe sur l'environnement des variables libres.

```
def plus(x) :
    def fermeture(y): return x+y
    return fermeture
>>> h = plus(5)
>>> h(4)
9
```

RÉCURSION SANS RÉCURSION

Définitions récursives

En général, Python n'accepte pas les définitions récursives

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

(Bien que cette définition ne semble pas poser de problème, puisqu'on sait représenter une telle valeur en mémoire)

Les définitions récursives ne semblent être possibles que pour les fonctions. Pour créer une valeur récursive *similaire* à `v`, on peut utiliser un effet de bord :

```
>>> v = [1,None]
>>> v[1] = v
>>> v[1][1][1]
[1, [...]]
```

Qu'est-ce qu'une fonction récursive ?

Une fonction semble être un objet mathématique qui prend un argument et renvoie un résultat.

Par exemple, la fonction $x \mapsto x + 1$ est représentée par la valeur `lambda x: x+1`

Qu'est-ce qu'une fonction récursive ?

Une fonction semble être un objet mathématique qui prend un argument et renvoie un résultat.

Par exemple, la fonction $x \mapsto x + 1$ est représentée par la valeur `lambda x: x+1`

Question : Mais qu'est-ce qu'une fonction récursive ?

Qu'est-ce qu'une fonction récursive ?

Une fonction semble être un objet mathématique qui prend un argument et renvoie un résultat.

Par exemple, la fonction $x \mapsto x + 1$ est représentée par la valeur `lambda x: x+1`

Question : Mais qu'est-ce qu'une fonction récursive ?

Réponse : C'est une fonction qui fait "appel à elle-même" !

Qu'est-ce qu'une fonction récursive ?

Une fonction semble être un objet mathématique qui prend un argument et renvoie un résultat.

Par exemple, la fonction $x \mapsto x + 1$ est représentée par la valeur `lambda x: x+1`

Question : Mais qu'est-ce qu'une fonction récursive ?

Réponse : C'est une fonction qui fait "appel à elle-même" !

Question : Mais comment faire "appel à soi-même" ?

Qu'est-ce qu'une fonction récursive ?

Une fonction semble être un objet mathématique qui prend un argument et renvoie un résultat.

Par exemple, la fonction $x \mapsto x + 1$ est représentée par la valeur `lambda x: x+1`

Question : Mais qu'est-ce qu'une fonction récursive ?

Réponse : C'est une fonction qui fait "appel à elle-même" !

Question : Mais comment faire "appel à soi-même" ?

Réponse : Hum... il faut que la fonction ait un nom

Qu'est-ce qu'une fonction récursive ?

Une fonction semble être un objet mathématique qui prend un argument et renvoie un résultat.

Par exemple, la fonction $x \mapsto x + 1$ est représentée par la valeur `lambda x: x+1`

Question : Mais qu'est-ce qu'une fonction récursive ?

Réponse : C'est une fonction qui fait "appel à elle-même" !

Question : Mais comment faire "appel à soi-même" ?

Réponse : Hum... il faut que la fonction ait un nom

Question : Donc le nom d'une fonction est important ?

Qu'est-ce qu'une fonction récursive ?

Une fonction semble être un objet mathématique qui prend un argument et renvoie un résultat.

Par exemple, la fonction $x \mapsto x + 1$ est représentée par la valeur `lambda x: x+1`

Question : Mais qu'est-ce qu'une fonction récursive ?

Réponse : C'est une fonction qui fait "appel à elle-même" !

Question : Mais comment faire "appel à soi-même" ?

Réponse : Hum... il faut que la fonction ait un nom

Question : Donc le nom d'une fonction est important ?

En fait, non !

La suite du calcul

Dans la définition de la fonction récursive `f` suivante :

```
def f(x):
    if x == 0:
        return 0
    else:
        return x + f(x-1)
```

l'expression `f(x-1)` représente la suite du calcul, c'est-à-dire ce qu'il reste encore à faire pour que l'appel à `f` puisse renvoyer un résultat.

Abstraire la suite du calcul

Amusons-nous à abstraire la suite du calcul. Pour cela, on définit la fonction `ff` suivante, qui prend un argument supplémentaire (une fonction) représentant la suite du calcul à effecter :

```
def ff(suite,x):
  if x == 0 :
    return 0
  else:
    return x + suite(      , x-1)
```

Abstraire la suite du calcul

Amusons-nous à abstraire la suite du calcul. Pour cela, on définit la fonction `ff` suivante, qui prend un argument supplémentaire (une fonction) représentant la suite du calcul à effecter :

```
def ff(suite,x):
  if x == 0 :
    return 0
  else:
    return x + suite(suite, x-1)
```

Cette suite peut aussi avoir besoin d'elle-même pour calculer ! !
C'est là que réside toute la subtilité de la récursion !

Abstraire la suite du calcul

Amusons-nous à abstraire la suite du calcul. Pour cela, on définit la fonction `ff` suivante, qui prend un argument supplémentaire (une fonction) représentant la suite du calcul à effecter :

```
def ff(suite,x):
  if x == 0 :
    return 0
  else:
    return x + suite(suite, x-1)
```

Cette suite peut aussi avoir besoin d'elle-même pour calculer ! !
C'est là que réside toute la subtilité de la récursion !

On notera bien que la fonction `ff` n'est pas récursive.

Récursion sans récursion

Finalement, la fonction `f` peut se définir comme une fonction qui appelle `ff` et qui utilise ff comme suite du calcul (ce qui nous rapproche de la définition de la récursivité).

```
>>> f = lambda x : ff(ff,x)
>>> f(5)
15
```

Récursion sans récursion

Finalement, la fonction `f` peut se définir comme une fonction qui appelle `ff` et qui utilise ff comme suite du calcul (ce qui nous rapproche de la définition de la récursivité).

```
>>> f = lambda x : ff(ff,x)
>>> f(5)
15
```

Mais il n'y a plus besoin de nommer une fonction pour calculer de manière récursive.

```
>>> f = lambda x: \
        (lambda f,y: 0 if y==0 else y + f(f,y-1)) \
        (lambda f,y: 0 if y==0 else y + f(f,y-1),x)
>>> f(5)
15
```

PROGRAMMATION AVEC ITÉRATEURS

Itérateurs

Un itérateur est une fonction qui permet de parcourir une structure de données et de réaliser un calcul.

L'utilisation d'un itérateur est nécessaire quand la structure de données est abstraite.

La programmation avec un itérateur évite de définir des fonctions récursives.

L'itérateur `filter`

L'itérateur `filter` est équivalent à l'opérateur de construction de listes pas compréhension.

```
list(filter(f,l)) == [ v for v in l if f(v)]
```

Par exemple,

```
>>> list(filter(lambda x: x%2==0,[10,2,31,42,54,67]))
[10, 2, 42, 54]
```

L'itérateur map

La sémantique de cet itérateur est définie par l'équation suivante :

```
list(map(f,[e1,e2,...,en])) == [f(e1),f(e2),...,f(en)]
```

Par exemple,

```
>>> list(map(lambda x:x*2, [1,2,3]))
[2, 4, 6]
```

L'itérateur reduce

Le schéma de calcul de cet itérateur est défini par les deux équations suivantes :

```
reduce(f,[ ],v) == v

reduce(f,[e1,e2, ...,en],v) ==
    f(...(f(f(v,e1),e2), ...),en)
```

Attention, ajouter la directive suivante pour utiliser `reduce` :

```
from functools import reduce
```

Une deuxième version s'applique à une liste non vide :

```
reduce(f,[e1,e2, ...,en]) ==
    reduce(f,[e2, ...,en],e1)
```

Exemples

La somme des éléments d'une liste d'entiers

```
def somme(l):
    return reduce(lambda s,x: s+x, l, 0)
```

Exemples

La somme des éléments d'une liste d'entiers

```
def somme(l):
    return reduce(lambda s,x: s+x, l, 0)
```

Le nombre d'éléments dans une liste de listes :

```
def nb_elem_list_list(l):
  return reduce (lambda acc,s: \
              reduce (lambda nb,x: 1+nb, s, acc), l, 0)
```

Exemple : les sous-listes

Calcul de la **liste des sous-listes** d'une liste :

```
def sous_listes(l):
 return reduce (lambda p,x: \
                  list(map(lambda k:k+[x],p))+p,l,[[]])
```

Exemple : les sous-listes

Calcul de la **liste des sous-listes** d'une liste :

```
def sous_listes(l):
 return reduce (lambda p,x: \
                  list(map(lambda k:k+[x],p))+p,l,[[]])
```

Exemple avec `sous_listes([1,2,3])`. Supposons (hypothèse de récurrence) que `p` soit la liste des sous-listes de `[1,2]` et `x == 3`.

```
p == [[1, 2], [2], [1], []]
```

Exemple : les sous-listes

Calcul de la **liste des sous-listes** d'une liste :

```
def sous_listes(l):
 return reduce (lambda p,x: \
                  list(map(lambda k:k+[x],p))+p,l,[[]])
```

Exemple avec `sous_listes([1,2,3])`. Supposons (hypothèse de récurrence) que `p` soit la liste des sous-listes de `[1,2]` et `x == 3`.

```
p == [[1, 2], [2], [1], []]
```

Le résultat de `list(map(lambda k:k+[x],p))` est

```
[[1, 2, 3], [2, 3], [1, 3], [3]]
```

Exemple : les sous-listes

Calcul de la **liste des sous-listes** d'une liste :

```
def sous_listes(l):
 return reduce (lambda p,x: \
                  list(map(lambda k:k+[x],p))+p,l,[[]])
```

Exemple avec `sous_listes([1,2,3])`. Supposons (hypothèse de récurrence) que `p` soit la liste des sous-listes de `[1,2]` et `x == 3`.

```
p == [[1, 2], [2], [1], []]
```

Le résultat de `list(map(lambda k:k+[x],p))` est

```
[[1, 2, 3], [2, 3], [1, 3], [3]]
```

Par définition, `p` contient aussi des sous-listes de `[1,2,3]` donc on l'ajoute au résultat :

```
[[1, 2, 3], [2, 3], [1, 3], [3]] + p
```

STRUCTURES DE DONNÉES IMMUABLES

Structures de données non modifiables

La programmation fonctionnelle se caractérise aussi par l'utilisation de structures de données immuables.

Il s'agit de structures de données, a priori quelconque (tableau, ensemble, dictionnaire, etc.) que l'on ne peut plus modifier une fois qu'elles sont construites.

Mais on peut néanmoins les utiliser, pour en consulter le contenu ou pour construire d'autres structures de données.

Interfaces

L'aspect immuable d'une structure se voit, non pas dans la manière dont elle est implantée, mais dans la signature des fonctions de son interface.

L'interface d'une structure immuable contiendra donc des fonctions pour :

- créer des structures (vide, à partir d'autres structures)
- ajouter des éléments
- fusionner des structures
- supprimer des éléments dans une stucture

Toutes ces fonctions doivent toujours renvoyer de nouvelles structures, sans jamais modifier les structures passées en arguments.

La transparence référentielle

La programmation fonctionnelle se caractérise par la propriété de transparence référentielle :

Quand on applique deux fois la même fonction aux mêmes arguments, alors on obtient le même résultat.

Partage

On pense souvent à tort que les structures de données immuables sont moins efficaces que les structures modifiables.

En réalité, on peut profiter de l'immuabilité d'une structure pour réaliser un maximum de partage quand on construit une nouvelle structure (pensez par exemple à l'ajout d'un élément dans un arbre)

Par exemple, comment vérifier efficacement (sans copies inefficaces) l'égalité suivante sur des ensembles, autrement qu'avec des structures immuables ?

$$A \cup B = (A \backslash B) \cup (B \backslash A) \cup (A \cap B)$$

Nombre variable d'arguments (1/2)

Il est possible de définir des fonctions pouvant accepter un nombre variable d'arguments. On utilise pour cela la notation `*params` pour déclarer les arguments de la fonction. Par exemple,

```
def somme(*params):
    cpt = 0
    for i in params:
        cpt = cpt + i
    return cpt

>>> somme(1,2,3,4)
10
>>> somme(1,2)
3
>>> somme()
0
```

Remarque : `params` correspond à un tuple et on accède à ses éléments avec `params[0]`, etc.

Nombre variable d'arguments (2/2)

Inversement, on peut utiliser la notation `*args` pour passer un tuple `args` à une fonction qui attend `len(args)` arguments.

```
def somme(a,b,c):
    return a+b+c

>>> l = (1,2,3)
>>> somme(*l)
6
```

Les décorateurs

Il s'agit d'un mécanisme fonctionel pour écrire des fonctions qui modifient d'autres fonctions

```
def deco(f):
  print('Execution du decorateur')
  def f_prime(*param):
      print('debut de la fonction modifiee')
      if param[1] == 0 :
          return 0
      else:
          return f(*param)
  return f_prime

@deco
def f(x,y):
   return x//y

>>> f(4,2)
2
>>> f(4,0)
0
```

PROGRAMMATION OBJETS EN PYTHON

Les classes

Le premier concept des langages à objets est celui de classe qui permet de regrouper dans une même entité les *données* et des *algorithmes* (sous forme de fonctions) sur ces données.

```
from math import sqrt

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

    def norme(self):
        return sqrt(self.x * self.x + self.y * self.y)
```

Le constructeur de la classe est la fonction `__init__(...)`

Cette fonction prend obligatoirement en premier argument l'objet (`self`) qu'elle doit initialiser

Remarque : Cette fonction ne renvoie rien, elle ne fait que des effets de bord

Objets

Les objets sont des *instances* particulières d'une classe.

Par exemple, la déclaration

```
v = Vecteur(10,4)
```

a pour effet de déclarer une nouvelle variable `v` dont la valeur est une nouvelle instance de la classe `Vecteur`

Le constructeur de la classe (la fonction `__init__(...)`) est appelé implicitement lors de la déclaration.

L'objet renvoyé (et initialisé par le constructeur) est alloué sur le tas.

Champs d'un objets

Les données contenues dans un objets sont stockées dans les champs (on dit également attributs ou variables d'instances) de l'objet.

Dans la classe `Vecteur`, les champs sont `x` et `y`.

Par exemple, on peut accéder aux champs de `v` et les modifier, avec la notation pointée

```
>>> print("(",v.x,",",v.y,")")
( 10 , 4 )
>>> v.x = 1
>>> v.y = 1
>>> print("(",v.x,",",v.y,")")
( 1 , 1 )
```

Encapsulation (1/2)

Un des intérêts de la programmation objet (en général) est l'encapsulation qui donne la possibilité d'interdire l'accès à certains champs d'un objet en les rendant invisibles à l'extérieur de la classe.

Cela permet par exemple de maintenir des invariants (par exemple que l'on manipule toujours des vecteurs normalisés).

Encapsulation (2/2)

Il n'y a pas vraiment de mécanisme pour déclarer des champs privés en Python, mais on peut *simuler* cela en ajoutant `__` (deux caractères soulignés) devant le nom du champ.

```
class Vecteur:
    def __init__(self, x, y):
        self.__x = x
        self.__y = y
>>> v = Vecteur(1,2)
>>> v.__x
Traceback (most recent call last):
  File "<pyshell#25>", line 1, in <module>
    v.__x
AttributeError: 'Vecteur' object has no attribute'__x'
```

Par exempl, le champ `__x` est toujours accessible à l'extérieur, mais son nom est automatiquement transformé en `_Vecteur__x`

```
>>> v._Vecteur__x
1
```

Méthodes

Les fonctions définies dans une classe sont appelées des méthodes

Une méthode définie dans une classe s'applique sur un objet de cette classe (ou qui *peut être vu comme* étant de cette classe) de la manière suivante :

```
>>> v.norme()
2.2360679774997979
```

Comme pour le constructeur de classe, une méthode prend comme premier argument l'objet sur lequel elle est appliquée. Le nom de cet objet est arbitraire, mais on utilise souvent self pour le désigner.

Variables statiques (ou de classe)

Une classe peut aussi contenir ses propres variables, appelées variables de classe. Ces variables sont liées à la classe et non aux instances de la classe.

```
class A:
    vitesse = 5
    def __init__(self, n):
        self.x = A.vitesse + n
```

La variable `vitesse` appartient à la classe `A` et on y accède en écrivant `A.vitesse`

Ces variables sont également modifiables.

```
>>> A.vitesse = 10
>>> p2 = A(6)
>>> p2.x
16
```

Méthodes statiques

Il est également possible de définir des méthodes appartenant à une classe. Il s'agit des méthodes statiques.

En Python, il suffit de définir une méthode en la précédant du décorateur `@staticmethod`

```
class B:
    @staticmethod
    def g(x):
        return x + 1

>>> B.g(5)
6
```

Remarque : une méthode statique ne s'applique pas à un objet, il ne faut donc pas ajouter l'argument self

Contrôle d'accès

En Python, il n'y a aucun contrôle d'accès aux éléments d'une classe.

Les attributs peuvent être librement lus et modifiés, il n'y a pas moyen de les marquer comme privés.

On peut non seulement modifier des attributs, mais aussi en ajouter et écraser des méthodes !

```
>>> v = Vecteur(4,2)
>>> v.z = 0                # nouvelle variable d'instance z
>>> v.norme = 'toto'       # on ecrase la methode norme
```

Héritage

Un autre concept important de la programmation Objet est celui d'héritage : une classe peut être définie comme héritant d'une ou plusieurs autres classes.

Les objets de la classe définie par héritage héritent de tous les champs et méthodes des classes héritées, auxquels ils peuvent ajouter de nouveaux champs ou de nouvelles méthodes.

Héritage simple

En Python, on peut définir un classe en héritant d'une autre classe, c'est l'héritage simple.

```
class A:
    v = 10
    def __init__(self,x):
        self.x = A.v + x

    def f(self,y):
        return self.x - y

class B(A):  # B herite de A
    def g(self,z):
        return z - self.x + self.f(z)

>>> p = B(7)
>>> p.g(100) + p.f(10)
7
```

Héritage simple et initialisation

L'initialisation des objets d'une classe définie par héritage se fait par un appel explicite au constructeur de la classe mère (ou *super classe*) à l'aide de la notation `super()`.

```
class B(A):
  def __init__(self,w):
      self.w = w
      super().__init__(w+5)
```

La fonction `super()` renvoie la classe parente de celle dans laquelle on se trouve. Donc `super().__init__(self,...)` appelle le constructeur de la classe parente de `self`.

L'appel au constructeur de la classe mère peut aussi se faire de la manière suivante :

```
class B(A):
  def __init__(self,w):
      self.w = w
      A.__init__(self,w+5)
```

Héritage multiple (1/1)

Une classe peut également hériter de plusieurs classes.

```
class A:
    def __init__(self,n): self.age = n
    def incr(self): self.age += 1
class B:
    def __init__(self,n): self.nom = n
    def affiche(self): print(self.nom, end='')

class C(A,B): # C herite de A et B
    def __init__(self,p,a):
        B.__init__(self,p)
        A.__init__(self,a)
    def anniversaire(self):
        self.incr()
        self.affiche()
        print(" a", self.age)
>>> p = C("Toto",10)
>>> p.anniversaire()
Toto a 11 ans
```

Héritage multiple (2/2)

Il convient de faire attention quand une classe hérite de plusieurs autres classes qu'il n'y ait pas conflits entre les noms de variables d'instances ou de méthodes.

En Python 3, l'ordre de recherche d'une méthode suit l'ordre spécifié à l'héritage

On peut connaître l'ordre de résolution avec la méthode `.mro()` (method résolution order) d'une classe.

```
class A:
      def f(self): print ("Dans A")
class B:
      def f(self): print ("Dans B")
class C(A, B): pass
>>> c.C()
>>> c.f()
Dans A
>>> C.mro()
[<class 'C'>,<class 'A'>,<class 'B'>,<class 'object'>]
```

Méthodes de classe

Python fournit un autre mécanisme de méthodes statiques à l'aide du décorateur `@classmethod`. Contrairement au décorateur `@staticmethod`, une méthode définie de cette manière est paramétrée par la classe sur laquelle elles est appelée.

```
class B:
    v = 10
    @staticmethod
    def g(x): return x + B.v

    @classmethod # C represente la classe appelante
    def h(C,x): return x + C.v

class D(B):
    v = 50

>>> B.g(5)
15
>>> D.h(5)
55
```

Finaliseurs

Il est également possible de définir un finaliseur pour une classe, c'est-à-dire une méthode qui est appelée quand un objet de cette classe est collecté par le GC

```
class B:
    def __del__(self):
        print('Finalisation')

def f():
    x = B()
    return

>>> f()
Finalisation
>>>
```

Objets : notions avancées

Dans la suite, nous allons aborder trois notions avancées de la programmation objet :

- Redéfinition
- Classe abstraite
- Sous-typage

Pour illustrer ces notions, nous utiliserons un exemple de modélisation d'objets graphiques (voir planche suivante)

Exemple

```
class Graphical:
    def __init__(self,x=0,y=0,w=0,h=0):
        self.x = x; self.y = y;
        self.width = w; self.heigth = h
    def move(self,dx,dy): self.x += dx; self.y +=dy
    def draw(self):
        # fonction qui ne fait rien
        return

class Rectangle(Graphical):
    def __init__(self,x1,y1,x2,y2):
        super().__init__((x1+x2)/2, (y1+y2)/2, abs(x1-
                                        x2), abs(y1-y2))
    def draw(self):
        print("Je dessine un rectangle")

>>> r = Rectangle(10,10,100,100)
>>> r.draw()
Je dessine un rectangle
>>> r.move(5,5)
```

Redéfinition (1/2)

Dans l'exemple précédent, on redéfinit (*overwriting*) dans la classe `Rectangle` la méthode `draw` de la classe `Graphical`.

De même, on peut définir une classe `Circle` qui hérite de `Graphical` et redéfinit aussi la méthode `draw`

```
class Circle(Graphical):
    def __init__(self,x,y,r):
        self.r = r
        super().__init__(x,y,2*r,2*r)
    def draw(self):
        print("Je dessine un cercle")
        return
```

Redéfinition (2/2)

L'intérêt de l'héritage et des redéfinitions s'illustre alors facilement avec l'exemple suivant :

```
def dessiner(t):
    for i in range(0,len(t)):
        t[i].draw()

>>> t = [Rectangle(10,10,100,100), Circle(50,50,10)]
>>> dessiner(t)
Je dessine un rectangle
Je dessine un cercle
```

Classes abstraites (1/2)

On peut remarquer qu'il n'a pas lieu de créer des instances de la classe `Graphical` : c'est ce qu'on appelle une classe abstraite.

En effet, certaines méthodes comme `draw` ne sont pas fournies et elles doivent être redéfinies dans les sous-classes.

On peut formaliser cela en utilisant les metaclass de Python.

```
from abc import ABCMeta, abstractmethod

class Graphical(metaclass=ABCMeta):

    def __init__(self,x=0,y=0,w=0,h=0):
        self.x = x; self.y = y;
        self.width = w; self.heigth = h

    def move(self,dx,dy): self.x += dx; self.y +=dy

    @abstractmethod
    def draw(self): pass
```

Classes abstraites (2/2)

Ainsi, il n'est pas possible de créer un objet en instantiant directement la classe `Graphical`.

```
>>> p = Graphical()
Traceback (most recent call last):
    p = Graphical()
TypeError:
  Can't instantiate abstract class Graphical with
                                  abstract methods draw
```

Il est alors obligatoire de redéfnir la méthode `draw` dans toutes les classes qui héritent de `Graphical`. Autrement, on obtient une erreur similaire pour les objets de ces nouvelles classes.

Typage

Les classes définissent de nouveaux types de données

Par exemple, lorsqu'on déclare

```
class A:
    pass

>>> p = A()
>>> type(p)
<class '__main__.A'>
```

Le type de p est `<class '__main__.A'>`

On peut également savoir si un objet appartient à une classe :

```
>>> isinstance(p,A)
True
```

Sous-Typage

La notion d'héritage s'accompagne d'une notion de sous-typage : un objet d'une classe `B` peut être vu comme un objet d'une classe `A`, si `B` hérite de `A`.

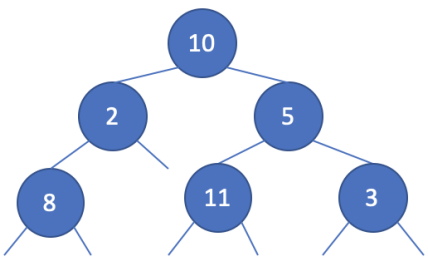
```
class A:
    def __init__(self,x):
        self.x = x

def f(p):
    assert(isinstance(p,A))
    return p.x + 1

class B(A):
    def __init__(self,y):
        super().__init__(y)

>>> p = B(10)
>>> f(p)
11
```

EXEMPLES DE STRUCTURES ARBORESCENTES DÉFINIES AVEC DES OBJETS

Les arbres binaires

Les arbres binaires avec des informations aux nœuds peuvent être définis avec une classe `Noeud` comme ci-dessous

```
class Noeud:
    def __init__(self,v,l,r):
        self.left = l
        self.value = v
        self.right = r
```



On peut créer l'arbre dessiné à gauche à l'aide de l'expression suivante :

```
a = Noeud(10,
      Noeud(2,Noeud(8,None,None),None),
      Noeud(5,Noeud(11,None,None),Noeud(3,None,None)))
```

Taille et hauteur d'un arbre binaire

La fonction `size` ci-dessous renvoie le nombre de nœuds d'un arbre binaire

```
def size(a):
    if a is None:
        return 0
    else:
        return 1 + size(a.left) + size(a.right)
```

La fonction `height` renvoie la longueur de la plus grande branche d'un arbre binaire

```
def height(a):
    if a is None:
        return 0
    else:
        return 1 + max(height(a.left), height(a.right))
```

Recherche dans un arbre binaire

La fonction `find(e,a)` recherche un élément `e` dans un arbre binaire `a`

```
def find(e, a):
    if a is None:
        return False
    else:
    return a.value == e or \
           find(e, a.left) or
           find(e, a.right)
```

- Le temps de recherche dans le pire des cas est en O(n)
- Il faut des hypothèses plus fortes sur la structure de l'arbre pour obtenir une meilleure complexité

Arbres binaires orientés objets (1/4)

On souhaite définir des arbres binaires comme des objets possédant des méthodes `size`, `find`, etc.

Comme il existe deux types d'arbres binaires (les feuilles et les nœuds), on commence par définir une classe abstraite `BinTree` qui regroupe l'interface de ces deux types d'arbre.

```
from abc import ABC, abstractmethod

class BinTree(ABC):
    def __init__(self):
        pass

    @abstractmethod
    def size(self):
        pass

    @abstractmethod
    def find(self,v):
        pass
```

Arbres binaires orientés objets (2/4)

Les feuilles d'un arbre sont définies à l'aide d'une classe `Leaf` qui hérite de `BinTree`

```
class Leaf(BinTree):
    def __init__(self):
        pass

    def size(self):
        return 0

    def find(self,_):
        return False
```

Les méthodes `size` et `find` sont simplement définies en intégrant le code relatif aux feuilles dans les fonctions définies précédemment

Arbres binaires orientés objets (3/4)

De la même manière, les nœuds d'un arbre sont définis à l'aide d'une classe `Node` qui hérite de `BinTree`

```
class Node(BinTree):
    def __init__(self,v,l,r):
        self.value = v
        self.left = l
        self.right = r

    def size(self):
        return 1 + self.left.size() + self.right.size()

    def find(self,v):
        return self.value == v or self.left.find(v) \
               or self.right.find(v)
```

Le code des deux méthodes `size` et `find` est également extrait des fonctions définies précédemment

Arbres binaires orientés objets (4/4)

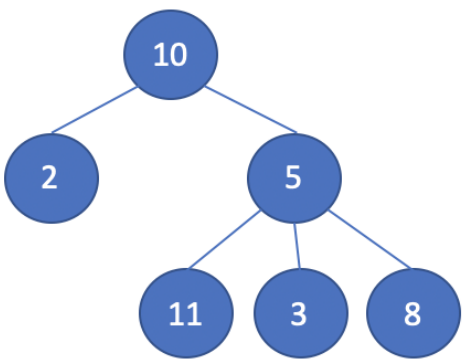
```
>>> t = Node(10,
         Node(2,Node(8,Leaf(),Leaf()),Leaf()),
         Node(5,Node(11,Leaf(),Leaf()),Node(3,Leaf(),
                                            Leaf())))
>>> t.size()
6
>>> t.find(11)
True
>>> t.find(42)
False
```

Arbres N-aires (1/2)

Les nœuds d'un arbre n-aire ont un nombre arbitraire de sous-arbres

On pourra par exemple représenter les nœuds d'un arbre binaire comme un objet de la classe suivante :

```
class N:
    def __init__(self,v,l):
        self.value = v
        self.childs = l
```



Ainsi, on peut créer l'arbre dessiné à gauche à l'aide de l'expression suivante :

```
a = N(10,[N(2,[]), N(5,[N(11,[]), N(3,[]), N(8,[])])])
```

Arbres N-aires (2/2)

La plupart des fonctions sur les arbres n-aires entremêlent les applications d'itérateurs (comme `map` ou `reduce`) et des appels récursifs de fonctions

```
    def size(self):
        return 1 + reduce (lambda acc, x: acc + x.size
                                        (), self.childs,0)

    def height(self):
        return 1 + reduce (lambda acc, x: max(acc,x.
                                        height()), self.
                                        childs, 0)

    def to_list(self):
        return reduce(lambda acc,a: acc+a.to_list(),
                                        self.childs, [self.
                                        value])
```

```
>>> a.to_list()
[10, 2, 5, 11, 3, 8]
```

PANDAS ET MATPLOTLIB

Voir cours de Kim Nguyen