DIU Enseigner l'informatique au lycée

Notes de cours

Sylvain Conchon

Sylvain.Conchon@lri.fr

Modularité

Notions introduites

- Modules et interfaces
- Encapsulation

Génie logiciel

Développer de grands programmes demande une certaine organisation, et en particulier un découpage des différents aspects du programme et des différentes tâches qui doivent être accomplies.

$\rightarrow$ programmer à plusieurs sur un même projet

$\rightarrow$ programmes de grande taille

Ces questions relevent du génie logiciel (GL)

Un des objectifs du GL : spécifier le rôle de chaque partie suffisamment précisément pour que chacune puisse ensuite être réalisée indépendamment des autres.

Modules

Pour développer du code à grande échelle il faut circonscrire et séparer proprement les différentes parties du programme.

Par exemple :

- → Séparer le code qui définit une structure de données du code qui utilise cette structure
- → Séparer l'interface graphique du cœur de l'application.
- → etc.

Chacun des morceaux obtenus peut être placé dans un fichier de code différent, appelé un module.

Modules en Python

Un module Python est simplement un fichier de code Python qui contient des définitions et des instructions

Un programme est découpé en plusieurs modules, les fonctionnalités définies dans un module pouvant être utilisées par d'autres.

Un module peut faire référence à d'autres modules. On dit qu'il dépend de ces autres modules.

Le nom d'un module défini par un fichier de code s'obtient simplement en retirant le suffixe `.py` au nom du fichier.

La commande `import`

La commande `import` permet d'utiliser les valeurs ou fonctions définies dans un module.

Par exemple, pour utiliser le module `module1` dans son propre fichier, on ajoute la commande suivante

```
import module1
```

Cette commande a pour effet d'ajouter l'identificateur `module1` dans l'environnement des variables globales.

Il est ensuite possible d'utiliser les fonctions ou valeurs définies dans ce fichier en utilisant la notation pointée.

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

Effets de la commande `import`

Les instructions du module importées sont exécutées au moment de l'import. Par exemple, dans l'exemple ci-dessus, si `module1.py` contient :

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

Alors, la valeur 42 sera affichée (effet de l'exécution de la commande `import`, puis la variable `y` sera affectée et enfin l'instruction `print(y)` sera exécutée.

(démo 1)

Import de modules (suite)

Il est possible d'importer une sélection de valeurs (fonctions) dans un module.

On peut également donner un autre nom au module importé.

Enfin, les noms des valeurs importées par un module cachent les valeurs de même nom importées précédemment.

```
from module2 import f1, f2
import module3 as m
from module4 import f2 # cache la fonction f2
                       # de module2
```

Interfaces

Pour chaque module, on distingue :

- sa réalisation, c'est-à-dire le code lui-même, et
- son interface, consistant en une énumération des fonctions définies dans le module qui sont destinées à être utilisées dans la réalisation d'autres modules, appelés *clients*

L'interface d'un module est liée à sa documentation, et doit notamment expliciter ce qu'un utilisateur a besoin de connaître des fonctions proposées : comment et pour quoi les utiliser.

Interface : exemple

Interface pour manipuler des ensembles

fonction	description
`cree()`	crée et renvoie un ensemble vide
`contient(`s`, `x`)`	renvoie `True` si et seulement si l'ensemble s contient x
`ajoute(`s`, `x`)`	ajoute x à l'ensemble s

Utilisation d'un module

Par exemple, on peut utiliser l'interface d'un module d'ensembles pour déterminer si un tableau contient des doublons.

```
from ensemble import cree, contient, ajoute

def contient_doublon(t):
    """le tableau t contient-il un doublon"""
    s = cree()
    for x in t:
        if contient(s, x) :
            return True
        ajoute(s, x)
    return False
```

Réalisation d'une interface

Un module réalise une interface quand il définit (au moins) toutes les fonctions promises dans l'interface.

Une même interface peut admettre plusieurs réalisations radicalement différentes.

Une réalisation possible (évidente) est la suivante :

```
def cree():
    return set()

def contient(s, x):
    return x in s

def ajoute(s, x):
    s.add(x)
```

Encapsulation

Pour réaliser une interface, un module peut contenir bien plus de fonctions ou de structures destinées à son usage interne.

En général, ces valeurs ≪ internes ≫(hors interface) ne doivent pas être utilisées par les modules clients. Elles sont souvent qualifiées de code privé.

On utilise le terme d'encapsulation pour signifier que ces valeurs sont (ou doivent être) enfermées dans le module, comme dans une boite hermétique.

Encapsulation en Python

En Python, l'auteur d'un module peut indiquer que certains valeurs sont privées en faisant commencer leur nom par le symbole _ (souligné)

Par convention, tous les autres éléments sont « publics » et doivent être compris comme appartenant à l'interface

Mais l'encapsulation en Python est une pure convention

Rien dans les mécanismes du langage n'empêche l'accès aux éléments privés, ni leur utilisation, ni leur modification

Tout au plus, la directive `from <module> import *` n'importe-t-elle pas automatiquement les noms commençant par _

Exceptions

Notions introduites

- Lever et rattraper des exceptions

Exceptions

Les programmes peuvent s'interrompre avec des messages d'erreurs variés.

```
>>> t = [1, 1, 2, 5, 14, 42, 132]
>>> t[12]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Ces erreurs sont appelées en programmation des exceptions

Elle correspondent à la détection (ici faite par l'interprète Python lui-même) d'un problème empêchant la bonne exécution du programme.

Lorsqu'une exception survient, l'exécution du programme est interrompue sur le champ.

Exceptions prédéfinies

Le tableau suivant redonne quelques exceptions courantes, observables en utilisant les structures de base de Python.

exception	contexte
`NameError`	accès à une variable inexistante
`IndexError`	accès à un indice invalide d'un tableau
`KeyError`	accès à une clé inexistante d'un dictionnaire
`ZeroDivisionError`	division par zéro
`TypeError`	opération appliquée à des valeurs incompatibles

Lever une exception

Il est possible de déclencher directement toutes ces exceptions (on dit lever une exception) avec l'opération `raise` de Python.

```
>>> raise IndexError('indice trop grand')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: indice trop grand
```

Cette opération s'écrit en faisant suivre le mot-clé `raise` du nom de l'exception à lever, lui-même suivi entre parenthèses d'une chaîne de caractères donnant des informations sur l'erreur signalée.

La pile d'appels

Le message affiché lorsqu'une exception interrompt un programme donne un aperçu de l'état de la pile d'appels au moment où l'exception a été levée.

On y voit donc quelle première fonction a appelé quelle deuxième fonction qui a, à son tour, appelé quelle autre fonction, etc., jusqu'à arriver au point où l'exception a été levée.

Il s'agit d'une information inestimable pour comprendre le contexte du problème et le corriger.

(démo 2)

Interfaces et exceptions

Les exceptions peuvent être utilisées dans les fonctions formant l'interface d'un module, pour signaler à un utilisateur du module toute utilisation incorrecte de ces fonctions.

L'interface mentionnera dans ce cas quelles exceptions spécifiques sont levées et dans quelles conditions.

Erreurs prévues et imprévues

Les exceptions levées par un programme peuvent avoir plusieurs causes.

- des erreurs imprévues du programme
  $\rightarrow$ dans ces conditions, interrompre l'exécution du programme est légitime.
- des erreurs qui s'inscrivent dans le fonctionnement normal du programme
  $\rightarrow$ elles correspondent à des situations connues, exceptionnelles mais possibles.

Exemple

Dans le code ci-dessous, il est tout à fait envisageable que la chaîne de caractères entrée par l'utilisateur ne représente pas un entier valide.

```
x = int(input("Entrer un jour"))
```

Dans ce cas, la fonction `int` lève une exception `ValueError` lorsque l'entier fourni n'est pas dans la plage de valeurs attendue.

$\rightarrow$ Une telle exception, qui correspond à un événement prévisible, n'est pas forcément fatale.

Rattraper des exceptions

Dans le programme précédent, on peut anticiper l'erreur et prévoir un comportement alternatif pour cette situation exceptionnelle.

Pour cela, il faut rattraper cette exception, c'est-à-dire l'intercepter avant que l'exécution du programme ne soit définitivement abandonnée, avec la construction suivante.

```
try:
    x = int(input("Entrer un jour"))
except ValueError:
    print("Entrer un entier entre 1 et 7")
```

Le mot-clé `try` suivi du symbole : (deux-points) introduit un premier bloc de code, puis le mot-clé `except` suivi du nom de l'exception et du symbole : précède un deuxième bloc de code. On qualifiera le premier bloc de normal et le second d'alternatif.

Alternatives multiples

Si l'on prévoit que plusieurs exceptions peuvent être rattrapées, il est possible d'écrire plusieurs blocs alternatifs, chacun associé à sa ligne `except`.

```
try:
    <bloc normal>
except Exception1:
    <bloc alternatif 1>
except Exception2:
    <bloc alternatif 2>
```

Seules les exceptions levées lors de l'exécution du bloc normal peuvent être rattrapées (les exceptions d'un bloc alternatif ne sont jamais rattrapées par les blocs alternatifs suivants).

Programmation orientée objet

Notions introduites

- définition de classes
- création et manipulation d'objets
- attributs et méthodes
- héritage, classes astraites, sous-typage

Le paradigme objet

Le premier concept de la programmation objet est celui de classe qui permet de regrouper dans un même « espace » données et algorithmes sur ces données.

Cette notion de classe permet à la fois de définir (et nommer) des structures de données composites, mais aussi de structurer le code d'un programme.

→ Les classes peuvent jouer le même rôle d'encapsulation que les modules (voir discussion plus loin). En fait, les deux techniques se complètent.

Classes et attributs

Une classe définit et nomme une structure de données qui vient s'ajouter aux structures de base du langage.

La structure définie par une classe peut regrouper plusieurs composantes de natures variées.

Chacune de ces composantes est appelée un attribut (on dit aussi un champ ou une propriété) et est dotée d'un nom.

Classes et attributs : exemple

Exemple, des triplets d'entiers représentant des temps mesurés en heures, minutes et secondes.

On appellera la structure correspondante `Chrono`.

```
class Chrono:
    """une classe pour représenter un temps mesuré en
       heures, minutes et secondes"""
    def __init__(self, h, m, s):
        self.heures = h
        self.minutes = m
        self.secondes = s
```

La définition d'une nouvelle classe est introduite par le mot-clé `class`, suivi du nom choisi pour la classe et d'un symbole : (deux-points). Le nom de la classe commence par une lettre majuscule (par convention).

Classes et attributs : exemple (suite)

```
class Chrono:
    """une classe pour représenter un temps mesuré en
       heures, minutes et secondes"""
    def __init__(self, h, m, s):
        self.heures = h
        self.minutes = m
        self.secondes = s
```

Cette classe contient la définition d'une fonction `__init__` (voir les détails plus loin)

Notons à présent sur sa forme :

- Elle possède un premier paramètre appelé `self`.
- Trois paramètres correspondant aux trois composantes du triplet.
- Trois instructions de la forme `self.`a `= ...` correspondant aux trois composantes et affectant à chaque attribut sa valeur.

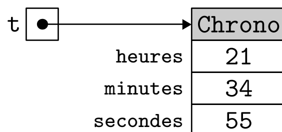
Création d'un objet

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

Par exemple, un objet correspondant à la structure `Chrono` peut être construit avec une expression de la forme `Chrono(`h`, `m`, `s`)`.

```
>>> t = Chrono(21, 34, 55)
```

Les objets sont alloués dans le tas mémoire.



Manipulation des attributs

On peut accéder aux attributs d'un objet `t` de la classe `Chrono` avec la notation `t.`a où a désigne le nom de l'attribut visé.

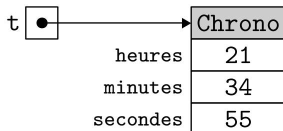
Les attributs, comme les cases d'un tableau, sont mutables en Python : on peut non seulement consulter leur valeur mais aussi la modifier.

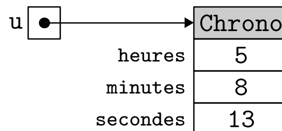
```
>>> t.secondes
55
>>> t.secondes = t.secondes + 1
>>> t.secondes
56
```

Attributs d'instance

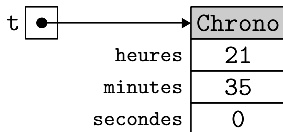
Bien que les noms des attributs soient attachés à une classe, chaque objet possède pour ses attributs des valeurs qui lui sont propres. On parle parfois aussi d'attributs d'instance.

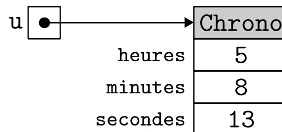
```
>>> t = Chrono(21, 34, 55)
>>> u = Chrono(5, 8, 13)
```






```
>>> t.secondes = 0
```





Erreurs : Attributs d'instance

Il n'est évidemment pas possible d'obtenir la valeur d'un attribut inexistant.

```
>>> t.x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Chrono' object has no attribute 'x'
```

Spécificités des attributs en Python

Rien n'empêche en Python d'affecter par mégarde une valeur à un attribut n'appartenant pas à la classe de l'objet.

```
>>> t.x = 89
>>> (t.heures, t.minutes, t.secondes, t.x)
(21, 34, 55, 89)
```

La structure des objets ne correspond pas tout à fait à la pratique usuelle de la programmation objet.

Les attributs ne sont pas réellement introduits au niveau de la classe. Plutôt, chaque affectation d'un attribut à un objet crée cet attribut pour cet objet particulier.

Python permet donc techniquement que deux objets d'une même classe possèdent des attributs n'ayant aucun rapport les uns avec les autres.

Attributs de classe (1/2)

Une classe peut également définir des attributs de classe, dont la valeur est attachée à la classe elle-même.

```
class Chrono:
    heure_max = 24
    ...
```

On peut consulter de tels attributs depuis n'importe quelle instance, ou depuis la classe elle-même.

```
>>> t = Chrono(21, 34, 55)
>>> (t.heure_max, Chrono.heure_max)
(24, 24)
```

Attributs de classe (2/2)

On peut également modifier cet attribut en y accédant via la classe elle-même pour que la modification soit perceptible par toutes les instances présentes ou futures.

```
>>> Chrono.heure_max = 12
>>> t.heure_max
12
```

En revanche, un tel attribut n'est pas destiné à être modifié depuis une instance (techniquement cela ne ferait que créer une variable d'instance du même nom, pour cette seule instance, qui serait donc décorrélée de l'attribut de classe).

Méthodes

Dans le paradigme de la programmation objet, la notion de classe est souvent associée à la notion d'encapsulation

Un programme manipulant un objet n'est pas censé accéder librement à la totalité de son contenu, une partie de ce contenu pouvant relever du détail d'implémentation.

La manipulation de l'objet passe donc de préférence par une interface constituée de fonctions dédiées, qui font partie de la définition de la classe et sont appelées les méthodes de cette classe.

Utilisation d'une méthode (1/2)

Les méthodes d'une classe servent à manipuler les objets de cette classe.

Chaque appel de méthode peut recevoir des paramètres.

Un appel s'applique à un objet de la classe concernée.

Par exemple, l'appel à une méthode `texte` s'appliquant au chronomètre `t` et renvoyant une chaîne de caractères décrivant le temps représenté par `t` est réalisé ainsi.

```
>>> t.texte()
'21h 34m 55s'
```

Cette notation pour l'appel de méthode utilise la même notation pointée que l'accès aux attributs de `t`, mais fait apparaître en plus une paire de parenthèses, comme pour l'appel d'une fonction sans paramètres.

Utilisation d'une méthode (2/2)

Lorsqu'une méthode dépend d'autres paramètres que cet objet principal `t`, ces autres paramètres apparaissent de la manière habituelle, entre les parenthèses et séparés par des virgules.

Par exemple, l'appel à une méthode `avance` faisant avancer le chronomètre `t` d'un certain nombre de secondes passé en paramètre s'écrit donc comme suit.

```
>>> t.avance(5)
>>> t.texte()
'21h 35m 0s'
```

On remarque qu'une méthode appliquée à l'objet `t` a la possibilité de modifier les attributs de cet objet.

Paramètres implicites et explicites

Lors d'un appel i`.m(`e_1`, ..., `e_n`)` à une méthode `m`, l'objet i est appelé le paramètre implicite et les paramètres e_1 à e_n les paramètres explicites.

Toutes les méthodes d'une classe attendent comme paramètre implicite un objet de cette classe.

Les paramètres explicites, en revanche, de même que l'éventuel résultat de la méthode, peuvent être des valeurs Python arbitraires.

Paramètres implicites et explicites : exemple

On peut imaginer dans notre classe `Chrono` une méthode `egale` s'appliquant à deux chronomètres (le paramètre implicite et un paramètre explicite) et testant l'égalité des temps représentés, et une méthode `clone` s'appliquant à un chronomètre `t` et renvoyant un nouveau chronomètre initialisé au même temps que `t`.

```
>>> u = t.clone()
>>> t.egale(u)
True
>>> t.avance(3)
>>> t.egale(u)
False
```

Définition d'une méthode

Une méthode d'une classe (et sa définition) peut être vue comme une fonction ordinaire, pouvant dépendre d'un nombre arbitraire de paramètres.

Mais une méthode doit nécessairement avoir pour premier paramètre un objet de cette classe (le paramètre implicite).

Les paramètres explicites 1 à n prennent les positions 2 à $n+1$.

En Python, par convention, ce premier paramètre est systématiquement appelé `self`.

Comme ce paramètre est un objet, on accède à ses attributs avec la notation `self.`a.

Définition d'une méthode (suite)

Ainsi, les méthodes `texte` et `avance` de la classe `Chrono` peuvent être définies de la manière suivante :

```
def texte(self):
    return (str(self.heures) + 'h '
            + str(self.minutes) + 'm '
            + str(self.secondes) + 's')

def avance(self, s):
    self.secondes += s
    # dépassement secondes
    self.minutes += self.secondes // 60
    self.secondes = self.secondes % 60
    # dépassement minutes
    self.heures += self.minutes // 60
    self.minutes = self.minutes % 60
```

Constructeur

La construction d'un nouvel objet avec une expression comme `Chrono(21, 34, 55)` déclenche deux choses :

1. la création de l'objet lui-même, gérée directement par l'interprète ou le compilateur du langage,
2. l'appel à une méthode spéciale chargée d'initialiser les valeurs des attributs. Cette méthode, appelée constructeur, est définie par le programmeur.

En Python, le constructeur est la méthode `__init__`.

La définition de cette méthode ne se distingue en rien de la définition d'une méthode ordinaire : son premier attribut est `self` et représente l'objet auquel elle s'applique, et ses autres paramètres sont les paramètres donnés explicitement lors de la construction.

La particularité de cette méthode est la manière dont elle est appelée, directement par l'interprète Python en réponse à une opération particulière.

Autres méthodes particulères en Python

méthode	appel	effet
`__str__(self)`	`str(t)`	renvoie une chaîne de caractères décrivant `t`
`__lt__(self, u)`	`t < u`	renvoie `True` si `t` est strictement plus petit que `u`
`__hash__(self)`	`hash(t)`	donne un code de hachage pour `t`, par exemple pour l'utiliser comme clé d'un dictionnaire `d`

Pour les collections.

méthode	appel	effet
`__len__(self)`	`len(t)`	renvoie un nombre entier définissant la taille de `t`
`__contains__(self, x)`	`x in t`	renvoie `True` si et seulement si `x` est dans la collection `t`
`__getitem__(self, i)`	`t[i]`	renvoie le `i`-ième élément de `t`

Égalité entre objets

Par défaut, la comparaison entre deux objets avec == ne considère pas comme égaux deux objets avec les mêmes valeurs pour chaque attribut : elle ne renvoie `True` que lorsqu'elle est appliquée deux fois au même objet, identifié par son adresse en mémoire.

Pour que cette comparaison caractérise les objets qui, sans être physiquement les mêmes, représentent la même valeur, il faut définir la méthode spéciale `__eq__(self, other)`.

On peut à cette occasion soit simplement comparer les valeurs de chaque attribut, soit appliquer un critère plus fin adapté à la classe représentée.

(démo 3)

Accès direct aux méthodes

Dans un style de programmation objet ordinaire, l'appel de méthode se fait exclusivement avec la notation t`.m(`e_1`, ..., `e_n`)`.

En Python, il reste toutefois possible d'accéder directement à une méthode `m` d'une classe `C` et de l'appeler comme une fonction ordinaire.

Il faut dans ce cas bien passer le paramètre implicite comme les autres :

`C.m(`t`, `e_1`, ..., `e_n`)`

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.

Exemple :

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

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

Ces variables sont également modifiables.

```
A.v = 10
p2 = A(6)
print(p2.x)
```

Méthodes statiques (ou de classe)

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

En Python, il suffit de définir une méthode sans ajouter l'argument self et de précéder la définition par `@staticmethod` comme ceci :

```
class B:

    @staticmethod
    def g(x):
        return x + 1

print (B.g(5))
```

Encapsulation

Un des intérêts de la programmation objet est l'encapsulation, c'est-à-dire la possibilité d'interdire l'accès à certains champs d'un objet en les rendant invisibles à l'extérieur de la classe.

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 Chrono:
    def __init__(self, h, m, s):
        self.__heures = h
        self.__minutes = m
        self.__secondes = s
```

Ces champs seront toujours accessibles à l'extérieur, mais leur nom sont automatiquement transformés en `_Chrono__heures`.

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 nouvelles méthodes.

Héritage simple

Certains langage de programmation ne permettent à une classe d'hériter que d'une seule 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):
    def g(self,z):
        return z - self.x

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

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)
```

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 (1/2)

Une classe peut également hériter de plusieurs classes. C'est l'héritage multiple.

```
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)

class C(A,B):
    def __init__(self,p,a):
        B.__init__(self,p)
        A.__init__(self,a)

    def anniversaire(self):
        self.incr()
        print("C'est l'anniversaire de "); self.affiche()
        print("Il a ", self.age, "ans")

p = C("Toto",10)
p.anniversaire()
```

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.

Python n'aide pas beaucoup dans ce cas : aucun messages d'erreur et on ne sait pas quelles variables ou méthodes seront utilisées `:-(`

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")
        return

r = Rectangle(10,10,100,100)
r.draw()
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.

```
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)
```

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`.

Par exemple, la déclaration `p = Graphical()` va provoquer l'erreur suivante :

```
Traceback (most recent call last):
  File "cours4.py", line 13, in <module>
    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:
    "classe A"

p = A()
print(type(p))
```

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

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

```
isinstance(p,A)
```

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)
print (f(p))
```

Récursivité

Notions introduites

- définitions récursives
- programmation avec fonctions récursives
- arbre d'appels
- modèle d'exécution et pile d'appels

Programmation à l'aide de fonctions récursives

Il s'agit à la fois d'un style de programmation mais également d'une technique pour définir des concepts et résoudre certains problèmes qu'il n'est parfois pas facile de traiter en programmant uniquement avec des boucles.

Le problème de la somme des n premiers entiers

Pour définir la somme des n premiers entiers, on a l'habitude d'écrire la formule suivante :

$$0 + 1 + 2 + \cdots + n \tag{1}$$

Bien que cette formule nous paraisse simple et intuitive, il n'est pas si évident de l'utiliser pour écrire en Python une fonction `somme(n)` qui renvoie la somme des `n` premiers entiers.

L'une des difficultés est de trouver un moyen de programmer la répétition des calculs qui est représentée par la notation $+ \cdots +$.

Une solution impérative (1/2)

On peut utiliser une boucle `for` pour parcourir tous les entiers `i` entre `0` et `n`, en s'aidant d'une variable locale `r` pour accumuler la somme des entiers de `0` à `i`.

```
def somme(n):
    r = 0
    for i in range(n + 1):
        r = r + i
    return r
```

Une solution impérative (2/2)

S'il n'est pas difficile de se convaincre que la fonction `somme(n)` calcule bien la somme des n premiers entiers, on peut néanmoins remarquer que ce code Python n'est pas directement lié à la formule (1).

En effet, il n'y a rien dans cette formule qui puisse laisser deviner qu'une variable intermédiaire `r` est nécessaire pour calculer cette somme.

Certains peuvent y voir l'art subtil de la programmation, d'autres peuvent se demander s'il ne serait pas possible de donner une définition mathématique plus précise à cette somme, à partir de laquelle il serait plus "simple" d'écrire un programme Python.

Une autre solution (1/2)

Une autre solution consiste à définir une fonction mathématique $somme(n)$ qui, pour tout entier naturel n, donne la somme des n premiers entiers de la manière suivante :

$$somme(n) = \begin{cases} 0 & \text{si } n = 0, \\ n + somme(n-1) & \text{si } n > 0. \end{cases}$$

Cette définition nous indique ce que vaut $somme(n)$ pour un entier n quelconque, selon que n soit égal à 0 ou strictement positif.

Ainsi, pour $n = 0$, la valeur de $somme(0)$ est simplement 0. Dans le cas où n est strictement positif, la valeur de $somme(n)$ est $n + somme(n-1)$.

Une autre solution (2/2)

Par exemple, voici ci-dessous les valeurs de $somme(n)$, pour n valant 0, 1, 2 et 3.

$$\begin{array}{lllllll} somme(0) & = & 0 & & & & \\ somme(1) & = & 1 + somme(0) & = & 1 + 0 & = & 1 \\ somme(2) & = & 2 + somme(1) & = & 2 + 1 & = & 3 \\ somme(3) & = & 3 + somme(2) & = & 3 + 3 & = & 6 \end{array}$$

Comme on peut le voir, la définition de $somme(n)$ dépend de la valeur de $somme(n-1)$.

Il s'agit là d'une définition récursive, c'est-à-dire d'une définition de fonction qui fait appel à elle-même.

Ainsi, pour connaître la valeur de $somme(n)$, il faut connaître la valeur de $somme(n-1)$, donc connaître la valeur de $somme(n-2)$, etc. Ceci jusqu'à la valeur de $somme(0)$ qui ne dépend de rien et vaut 0. La valeur de $somme(n)$ s'obtient en ajoutant toutes ces valeurs.

Une autre solution (3/3)

L'intérêt de cette définition récursive de la fonction $somme(n)$ est qu'elle est directement calculable, c'est-à-dire exécutable par un ordinateur.

En particulier, cette définition est directement programmable en Python, comme le montre le code ci-dessous.

```
def somme(n):
    if n == 0:
        return 0
    else:
        return n + somme(n - 1)
```

Évaluation de la fonction `somme`

L'évaluation de l'appel à `somme(3)` peut se représenter à l'aide d'un arbre d'appels de la manière suivante, où on indique uniquement pour chaque appel à `somme(n)` l'instruction qui est exécutée après le test `n == 0` de la conditionnelle.

```
somme(3) = return 3 +  somme(2)
                          |
                       return 2 +  somme(1)
                                      |
                                   return 1 +  somme(0)
                                                  |
                                               return 0
```

Évaluation de la fonction `somme`

L'évaluation de l'appel à `somme(3)` peut se représenter à l'aide d'un arbre d'appels de la manière suivante, où on indique uniquement pour chaque appel à `somme(n)` l'instruction qui est exécutée après le test `n == 0` de la conditionnelle.

```
somme(3) = return 3 +  somme(2)
                         |
                       return 2 +  somme(1)
                                     |
                                   return 1 + 0
```

Évaluation de la fonction `somme`

L'évaluation de l'appel à `somme(3)` peut se représenter à l'aide d'un arbre d'appels de la manière suivante, où on indique uniquement pour chaque appel à `somme(n)` l'instruction qui est exécutée après le test `n == 0` de la conditionnelle.

```
somme(3) = return 3 +  somme(2)
                         |
                       return 2 + 1
```

Évaluation de la fonction `somme`

L'évaluation de l'appel à `somme(3)` peut se représenter à l'aide d'un arbre d'appels de la manière suivante, où on indique uniquement pour chaque appel à `somme(n)` l'instruction qui est exécutée après le test `n == 0` de la conditionnelle.

`somme(3)` = `return 3 + 3`

Une notation ambiguë

La formule (1) est non seulement éloignée du programme qui la calcule, elle est également ambiguë quant à la spécification de son résultat.

À lire cette formule à la lettre, comment savoir si la somme des premiers entiers pour $n = 2$ est $0 + 1 + 2$ ou $0 + 1 + 2 + 2$?

Si la réponse à cette question peut sembler évidente, elle ne l'est que parce que nous avons l'habitude de la signification de $+ \cdots +$ et savons que les entiers (0, 1 et 2) dans cette formule sont déjà des instances de n.

De manière générale, la programmation imposant la précision, nous sommes souvent amenés comme ici à considérer avec une rigueur nouvelle certaines choses « évidentes ».

Formulations récursives

Une formulation récursive d'une fonction est toujours constituée de plusieurs cas, parmi lesquels on distingue des cas de base et des cas récursifs du calcul.

Les cas récursifs sont ceux qui renvoient à la fonction en train d'être définie ($somme(n) = n + somme(n-1)$ si $n > 0$).

Les cas de base de la définition sont à l'inverse ceux pour lesquels on peut obtenir le résultat sans avoir recours à la fonction définie elle-même ($somme(n) = 0$ si $n = 0$).

$\rightarrow$ Ces cas de base sont habituellement les cas de valeurs particulières pour lesquelles il est facile de déterminer le résultat.

Un deuxième exemple : la fonction puissance (1/3)

L'opération de puissance n-ième d'un nombre x s'écrit habituellement de la manière suivante :

$$x^n = \underbrace{x \times \cdots \times x}_{n \text{ fois}}$$

avec, par convention, que la puissance de x pour $n = 0$ vaut 1.

Un deuxième exemple : la fonction puissance (2/3)

Pour écrire une version récursive de x^n, on va définir une fonction $puissance(x, n)$ en commençant par chercher les cas de base à cette opération.

Ici, le cas de base évident est celui pour $n = 0$. On écrira donc la définition (partielle) suivante :

$$puissance(x, n) = \begin{cases} 1 & \text{si } n = 0, \\ ? & \text{si } n > 0. \end{cases}$$

Un deuxième exemple : la fonction puissance (3/3)

Pour définir la valeur de $puissance(x, n)$ pour un entier n strictement positif, on suppose que l'on connaît le résultat de x à la puissance $n - 1$, c'est-à-dire la valeur de $puissance(x, n - 1)$.

Dans ce cas, $puissance(x, n)$ peut simplement être définie par $x \times puissance(x, n - 1)$.

Au final, on obtient donc la définition suivante :

$$puissance(x, n) = \begin{cases} 1 & \text{si } n = 0, \\ x \times puissance(x, n-1) & \text{si } n > 0. \end{cases}$$

Faire confiance à la récursion

Pour faciliter l'écriture des cas récursifs, il est très important de supposer que les appels récursifs donnent les bons résultats pour les valeurs sur lesquelles ils opèrent, sans chercher à "construire" dans sa tête l'arbre des appels pour se convaincre du bien fondé de la définition.

Définitions récursives plus riches

Toute formulation récursive d'une fonction possède au moins un cas de base et un cas récursif.

Ceci étant posé, une grande variété de formes est possible.

Cas de base multiples

La définition de la fonction $puissance(x, n)$ n'est pas unique.

On peut par exemple identifier deux cas de base "faciles", celui pour $n = 0$ mais également celui pour $n = 1$ avec $puissance(x, 1) = x$.

Ce deuxième cas de base a l'avantage d'éviter de faire la multiplication (inutile) $x \times 1$ de la définition précédente.

$$puissance(x, n) = \begin{cases} 1 & \text{si } n = 0, \\ x & \text{si } n = 1, \\ x \times puissance(x, n-1) & \text{si } n > 1. \end{cases}$$

Cas récursifs multiples

Il est possible de définir une fonction avec plusieurs cas récursifs.

Par exemple, on peut donner une autre définition pour $puissance(x, n)$ en distinguant deux cas récursifs selon la parité de n.

En effet, si n est pair, on a alors $x^n = \left(x^{n/2}\right)^2$. De même, si n est impair, on a alors $x^n = x \times \left(x^{(n-1)/2}\right)^2$, où l'opération de division est supposée ici être la division entière.

$$puissance(x, n) = \begin{cases} 1 & \text{si } n = 0, \\ carre(puissance(x, n/2)) & \text{si } n \geq 1 \text{ et } n \text{ est pair}, \\ x \times carre(puissance(x, (n-1)/2)) & \text{si } n \geq 1 \text{ et } n \text{ est impair}. \end{cases}$$

Double récursion

Les expressions qui définissent une fonction peuvent aussi dépendre de plusieurs appels à la fonction en cours de définition.

Par exemple, la fonction $fib(n)$ est définie récursivement, pour tout entier naturel n, de la manière suivante :

$$fib(n) = \begin{cases} 0 & \text{si } n = 0, \\ 1 & \text{si } n = 1, \\ fib(n-2) + fib(n-1) & \text{si } n > 1. \end{cases}$$

Voici par exemple les premières valeurs de cette fonction.

$$\begin{array}{lllllll} fib(0) & = & 0 & & & & \\ fib(1) & = & 1 & & & & \\ fib(2) & = & fib(0) + fib(1) & = & 0+1 & = & 1 \\ fib(3) & = & fib(1) + fib(2) & = & 1+1 & = & 2 \\ fib(4) & = & fib(2) + fib(3) & = & 1+2 & = & 3 \\ fib(5) & = & fib(3) + fib(4) & = & 2+3 & = & 5 \\ \ldots & & & & & & \end{array}$$

Récursion imbriquée

Les occurrences de la fonction en cours de définition peuvent également être imbriquées.

Par exemple, la fonction $f_{91}(n)$ ci-dessous est définie avec deux occurrences imbriquées, de la manière suivante :

$$f_{91}(n) = \begin{cases} n - 10 & \text{si } n > 100, \\ f_{91}(f_{91}(n+11)) & \text{si } n \leq 100. \end{cases}$$

Voici par exemple la valeur de $f_{91}(99)$:

$$\begin{array}{rcll} f_{91}(99) & = & f_{91}(f_{91}(110)) & \text{puisque } 99 \leq 100, \\ & = & f_{91}(100) & \text{puisque } 110 > 100, \\ & = & f_{91}(f_{91}(111)) & \text{puisque } 100 \leq 100, \\ & = & f_{91}(101) & \text{puisque } 111 > 100, \\ & = & 91 & \text{puisque } 101 > 100. \end{array}$$

Récursion mutuelle

Il est également possible, et parfois nécessaire, de définir plusieurs fonctions récursives en même temps, quand ces fonctions font référence les unes aux autres.

On parle alors de définitions récursives mutuelles.

Par exemple, les fonctions $a(n)$ et $b(n)$ ci-dessous sont définies par récursion mutuelle de la manière suivante :

$$a(n) = \begin{cases} 1 & \text{si } n = 0, \\ n - b(a(n-1)) & \text{si } n > 0. \end{cases}$$
$$b(n) = \begin{cases} 0 & \text{si } n = 0, \\ n - a(b(n-1)) & \text{si } n > 0. \end{cases}$$

Définitions récursives bien formées

Il est important de respecter quelques règles élémentaires lorsqu'on écrit une définition récursive.

- (R1) Il faut s'assurer que la récursion va bien se terminer, c'est-à-dire que l'on va finir par "retomber" sur un cas de base de la définition.
- (R2) Il faut que les valeurs utilisées pour appeler la fonction soient toujours dans le domaine de la fonction.
- (R3)Il convient de vérifier qu'il y a bien une définition pour toutes les valeurs du domaine.

Règle (R1)

La fonction $f(n)$ ci-dessous définie de la manière suivante :

$$f(n) = \begin{cases} 1 & \text{si } n = 0, \\ n + f(n+1) & \text{si } n > 0. \end{cases}$$

est incorrecte car la valeur de $f(n)$, pour tout n strictement positif, ne permet pas d'atteindre le cas de base pour $n = 0$.

Par exemple, la valeur de $f(1)$ est :

$$f(1) = 1 + f(2) = 1 + 2 + f(3) = \ldots$$

Règle (R2)

La définition récursive de la fonction $g(n)$ s'applique uniquement à des entiers naturels.

$$g(n) = \begin{cases} 1 & \text{si } n = 0, \\ n + g(n-2) & \text{si } n > 0. \end{cases}$$

Cette définition n'est pas correcte car, par exemple, la valeur de $g(1)$ vaut

$$g(1) \quad = \quad 1 + g(-1)$$

mais $g(-1)$ n'a pas de sens puisque cette fonction ne s'applique qu'à des entiers naturels.

(Cette définition peut conduire à une exécution infinie ou à une erreur, selon la manière dont elle est écrite en Python)

Règle (R3)

La fonction $h(n)$ s'applique à des entiers naturels et est définie de la manière suivante :

$$h(n) = \begin{cases} 1 & \text{si } n = 0, \\ n + h(n-1) & \text{si } n > 1. \end{cases}$$

Cette définition est incorrecte, puisqu'une valeur est oubliée ($h(1)$).

Définition récursive de structures de données

Les techniques de définition récursive peuvent s'appliquer à toute une variété d'objet, et pas seulement à la définition de fonctions.

Vous verrez un peu plus tard des structures de données définies récursivement.

Programmer avec des fonctions récursives

Une fois que l'on dispose d'une définition récursive pour une fonction, il est en général assez facile de la programmer en Python.

Il faut néanmoins faire attention à deux points importants :

- Vérifier que le domaine mathématique d'une fonction est bien le même que l'ensemble des valeurs du type Python avec lesquelles elle sera appelée.
- Le choix d'une définition récursive plutôt qu'une autre peut dépendre du modèle d'exécution des fonctions récursives, en particulier quand il s'agit de prendre en compte des contraintes d'efficacité.

Domaine mathématique *vs.* type de données (1/4)

Le code de la fonction `somme(n)` ne se comporte pas exactement comme la fonction mathématique $somme(n)$

$\rightarrow$ La fonction mathématique est uniquement définie pour des entiers naturels, alors que la fonction `somme(n)` peut être appelée avec un entier Python arbitraire, qui peut être une valeur négative.

Par exemple, l'appel `somme(-1)` ne provoque aucune erreur immédiate, mais il implique un appel à `somme(-2)`, qui déclenche un appel à `somme(-3)`, etc.

(qui va finir par provoquer une erreur à l'exécution).

Domaine mathématique *vs.* type de données (2/4)

Pour éviter ce comportement, il y a plusieurs possibilités.

La première est de changer le test `n == 0` par `n <= 0`.

Cette solution a l'avantage de garantir la terminaison de la fonction, mais elle modifie la spécification de la fonction en renvoyant, de manière arbitraire, la valeur 0 pour chaque appel sur un nombre négatif.

Domaine mathématique *vs.* type de données (3/4)

Une autre solution est de restreindre les appels à la fonction `somme(n)` aux entiers positifs ou nuls.

Pour cela, on peut utiliser une instruction `assert`

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

De cette manière, une erreur sera déclenchée pour tout appel à `somme(n)` avec `n < 0`.

Bien que cette solution soit correcte, elle n'est pas encore complètement satisfaisante : pour tout appel `somme(n)` avec `n >= 0`, chaque appel récursif commencera par faire le test associé à l'instruction `assert`, alors que chaque valeur de `n` sera nécessairement positive.

Domaine mathématique *vs.* type de données (4/4)

Une solution pour éviter ces tests inutiles est de définir deux fonctions.

La première, `somme_bis(n)`, implémente la définition récursive de la fonction mathématique $somme(n)$ sans vérifier son argument.

```
def somme_bis(n):
    if n == 0:
        return 0
    else:
        return n + somme_bis(n - 1)
```

La seconde, `somme(n)`, est la fonction principale qui ne fait que vérifier (une et une seule fois) que son argument `n` est positif puis, si c'est le cas, elle appelle la fonction `somme_bis(n)`.

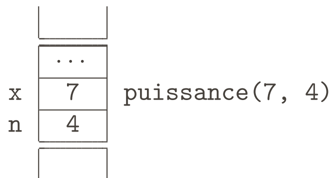
```
def somme(n):
    assert n >= 0
    return somme_bis(n)
```

Modèle d'exécution (1/3)

L'espace mémoire d'un programme est organisée sous forme d'une pile où sont stockés les contextes d'exécution de chaque appel de fonction.

```
def puissance(x, n):
    if n == 0:
        return 1
    else:
        return x * puissance(x, n - 1)
```

Par exemple, l'organisation de la mémoire pour l'appel à `puissance(7,4)` est :



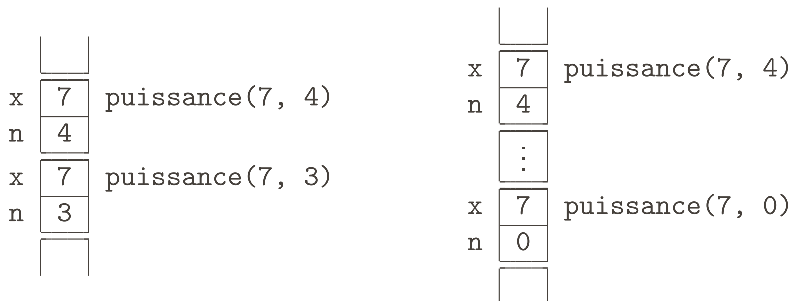
Modèle d'exécution (2/3)

Le calcul récursif de `puissance(7,4)` va engendrer une suite d'appels en cascade à la fonction `puissance`.

```
puissance(7,4) =
 return 7 * puissance(7,3)
                |
            return 7 * puissance(7,2)
                           |
                       return 7 * puissance(7,1)
                                      |
                                  return 7 * puissance(7,0)
                                                 |
                                             return 1
```

Modèle d'exécution (3/3)

Un environnement d'exécution va être alloué sur la pile pour chaque appel..



Juste après le dernier appel à `puissance(7,0)`, la pile contient donc les environnements d'exécution pour les cinq appels (schéma de droite).

Débordement de pile

Python limite explicitement le nombre d'appels récursifs dans une fonction à 1000 appels récursifs.

Au delà, l'interpréteur va lever l'exception `RecursionError` et afficher le message d'erreur suivant :

```
RecursionError: maximum recursion depth exceeded.
```

Cette limite, fixée à 1000 appels récursifs, est une valeur par défaut qu'il est possible de modifier en utilisant la fonction `setrecursionlimit` disponible dans le module `sys`.

Par exemple, pour passer cette limite à 2000 appels maximum, on exécutera le code Python suivant :

```
import sys
sys.setrecursionlimit(2000)
```

Un tel changement reste cependant dérisoire lorsque l'on a une définition récursive qui, par nature, effectue un très grand nombre d'appels emboîtés.

Une autre définition

On peut cependant utiliser une autre définition de la fonction mathématique $puissance(x, n)$ qui réduit drastiquement le nombre d'appels récursifs emboîtés.

On peut le faire avec deux cas récursifs qui distinguent la parité de n et deux cas de base ($n = 0$ et $n = 1$)

$$puissance(x, n) = \begin{cases} 1 & \text{si } n = 0, \\ x & \text{si } n = 1, \\ carre(puissance(x, n/2)) & \text{si } n > 1 \text{ et } n \text{ est pair}, \\ x \times carre(puissance(x, (n-1)/2)) & \text{si } n > 1 \text{ et } n \text{ est impair}. \end{cases}$$

Maîtriser la complexité

Cette fonction récursive a en effet l'avantage de diminuer largement le nombre d'appels récursifs.

Pour s'en convaincre, prenons l'exemple de l'appel `puissance(7, 28)`. L'arbre des appels (simplifié) représenté ci-dessous montre que seuls quatre appels récursifs sont nécessaires pour effectuer le calcul.

```
puissance(7, 28) =
 r = puissance(7, 14)···return r * r
  r = puissance(7, 7)···return r * r
   r = puissance(7, 3)···return 7 * r * r
    r = puissance(7, 1)···return 7 * r * r
     return 7
```

D'une manière générale, il faut $1 + \lfloor log_2(n) \rfloor$ appels pour calculer `puissance(x, n)` avec cette définition, c'est-à-dire un appel initial et $\lfloor log_2(n) \rfloor$ appels récursifs. Ainsi, le calcul de `puissance(x, 1000)` ne nécessite que $1 + \lfloor log_2(1000) \rfloor = 10$ appels.

La récursion dans d'autres langages

Certains langages de programmation, plus spécialisés que Python dans l'écriture de fonctions récursives, savent dans certains cas éviter de placer de trop nombreux environnements d'appel dans la pile.

Cela leur permet dans les cas en question de s'affranchir de toute limite relative au nombre d'appels emboîtés. C'est le cas notamment des langages fonctionnels.