Aide Mémoire Python

Petit guide de survie à l'attention des étudiants de CentraleSupélec.
Édition 2015

Python 3.4

CentraleSupélec

Marc-Antoine Weisser

Variables et Affectations

Une variable est une étiquette liée à un objet en mémoire. Elle n'a pas de *type* et ne nécessite pas de déclaration mais doit être affectée à un objet avant toute utilisation. Son nom ne peut contenir que les caractères `azAZ09_` et ne peut pas débuter par un chiffre. On peut détruire une variable avec le mot clef **`del`**.

```
variable = value                #affectation simple
v1, v2, v3 = val1, val2, val3   #affectation multiple
del v1                          #destruction de v1
```

`None` est une constante pour représenter ce qui n'a pas de valeur, pour autant, affecter **`None`** à une variable revient bien à lui donner une valeur.

On obtient le type d'un objet `o` avec la fonction `type(o)` et son identité avec `id(o)`. C'est un entier unique associé à tout objet : son adresse en mémoire.

Type Booléen

`True` et **`False`** sont les littéraux booléens. Quelque soit son type, une valeur peut être comparé à un booléen. Les valeurs **`None`**, **`False`**, `0`, `0.0`, `0j`, `''`, `()`, `[]` et `{}` sont considérées comme fausses. Toute autre est vraie.

Opérations de logique par ordre décroissant de priorité, **`or`** *et* **`and`** *sont coupe-circuits.*

Opérateurs	Description
`x or y`	si `x` est faux alors `y`, sinon `x`
`x and y`	si `x` est faux alors `x`, sinon `y`
`not x`	si `x` est faux alors **`True`**, sinon **`False`**

Types Numériques

Description des littéraux des trois types numériques de Python.

Type	Description	Litéraux	
`int`	Entiers signés sur 32 bits	`1,-26, 2085`	
`float`	Nombres à virgule flottante	`123.4`	`1.234e2`
`complex`	Nombres complexes	`1+1j`	`1e0+1e0j`

Les opérateurs numériques.

	Opérateurs	Description
Arithmétique	`x+y, x-y, x*y`	Somme, différence et produit
	`x/y`	Division réelle de `x` par `y`
	`x//y`	Partie entière du quotient de `x` par `y`
	`x**y, pow(x,y)`	`x` puissance `y`
	`abs(x)`	Valeur absolue de `x`
	`round(x)`	Arrondi : `round(3.57)` ⇒ `3.6`
	`x%y`	Reste de la division de `x` par `y`
Binaire	`x&y, x|y, x^y`	Opérations bit à bit (et/ou/xor) sur des **`int`**
	`x<<y, x>>y`	Décalage de bits non signés sur des **`int`**
	`~x`	Complément bit à bit sur des **`int`**

*Tous les opérateurs binaires (+, -, *, /, //, **, &, |, ^) peuvent être utilisés dans une affectation raccourcie.*

```
a+=b  ⇔  a=a+b
a**=b ⇔  a=a**b
a|=b  ⇔  a=a|b
```

Logique

Opérations de comparaisons. Le type de chaque opérande est quelconque. Le résultat est un booléen. Les opérations sont combinables : `12 <= x < 20`

Opérateurs	Description
`<, <=, >, >=`	$<, \leq, >, \geq$
`==, !=`	Égalité et inégalité de valeurs.
`is, is not`	Identité et différence d'objet. `a is b` ⇔ `id(a) == id(b)`

Description Générale

Les séquences sont des suites ordonnées d'objets. Les principales sont : `tuple`, `list`, `range` et `str`. Seuls les séquences de type `list` sont *mutables* : on peut les modifier. Les séquences sont des objets *itérables* (voir la *boucle for*), on peut accéder aux éléments à partir de leurs indices commençant à `0`.

Les quatre principales séquences.

Type	Description	Seq. Vide	Exemple
`tuple`	Séquence non-muables, types quelconques	`()`	`(1, 3.4, None)`
`str`	Séquence non-mutable de caractères	`""`, `''`	`"string", 'cc'`
`list`	Séquence mutable, types quelconques.	`[]`	`[2, 4, 6, 8]`
`range`	Séquence non-mutable d'entiers		`range(2,10,2)`

Opérations Communes aux Séquences

Sauf exception, ces opérations sont communes à toutes les séquences.

Opérations	Description
`x in s`	**`True`** si `s` contient `x`
`x not in s`	**`False`** si `s` ne contient pas `x`
`s+t`	Concaténation de `s` et `t`. Interdit pour `range`
`s*n, n*s`	`n` copies superficielles de `s` concaténées. Interdit pour `range`
`s[i]`	Élément d'indice `i`. Les indices débutent à `0`. Un indice négatif débute depuis la fin : `s[-1]` est le dernier élément.
`s[i:j]`	Slice : copie superficielle de la sous-séquence de l'indice `i` à `j`
`s[i:j:k]`	Slice de l'indice `i` à `j` avec un pas de `k`
`len(s)`	Longueur de la séquence `s`
`min(s), max(s)`	Valeur minimum et maximum de la séquence `s`
`sum(s)`	Somme les éléments contenus dans `s`.
`s.index(x)`	Indice de la première occurrence de `x` dans `s`
`s.count(x)`	Nombre d'occurrences de `x` dans `s`

La notation en slice permet d'omettre certaines valeurs.

```
s[:b]  #de l'indice 0 jusqu'à b
s[a:]  #de l'indice a jusqu'à la fin
s[:]   #toute la séquence (utile pour dupliquer une seq)
s[::p] #tous les p termes de la séquence du début à la fin
s[-1:0:-1] #la séquence renversée
```

Opérations sur les Séquences Mutables (list)

Ces opérations sont communes aux séquences mutables dont le type list.

Opérations	Description
`s[i]=x`	Replace la valeur d'indice `i` par `x`
`s[i:j]=t`	Replace le slice de `i` à `j` par les valeurs de l'itérable `t`.
`del` `s[i:j]`	Équivalent à `s[i:j]=[]`
`s[i:j:k]=t`	Remplace les objets de `s[i:j:k]` par ceux de `t` (même taille).
`del` `s[i:j:k]`	Supprime les valeurs de `s[i:j:k]`
`s.append(x)`	Ajoute l'élément `x` en fin de la séquence `s`
`s.clear()`	Vide la séquence `s`
`s.copy()`	Crée une copie superficielle de la séquence `s`
`s.extent(t)`	Ajoute les éléments de l'itérable `t` à la fin de la séquence `s`
`s.insert(i,x)`	Insère l'élément `x` à l'indice `i` (décalage de la fin de la séquence `s`)
`s.pop(i)`	Renvoie et supprime l'élément d'indice `i` de la séquence `s`
`s.remove(x)`	Retire le premier élément tel que `s[i]==x`
`s.reverse()`	Renverse l'ordre des éléments de la séquence `s`
`s.sort()`	Trie les éléments de `s`. Uniquement disponible pour le type list.

Chaînes de Caractères

Description Générale

Il existe plusieurs notation pour littéraux de type chaîne de caractères (`str`). Elles sont toutes équivalentes : `"Guillemets"`, `'Apostrophes'`, `"""Triple guillemets"""` et `'''Triple apostrophes'''`. Les notations triples peuvent contenir des retour à la ligne, les autres doivent utiliser le caractère échappé : `\n`. Précéder une chaîne de `r` annule l'échappement des caractères. Ainsi `r"\\"` donnera la chaîne : `\\` et non pas `\`.

Principaux caractères échapés

Code	Description
`\n`	Retour à la ligne
`\t`	Tabulation
`\a`	Bip

Code	Description
`\'`	Apostrophe
`\"`	Guillemets
`\N{name}`	Unicode *name*

Code	Description
`\\`	Backslash
`\b`	Backspace
`\ooo \xhh`	octal et hexa

Fonctions
`s.capitalize()` - Renvoie une chaîne avec le premier caractère majuscule et les autres en minuscule.
`s.endswith(suffix [,start [, end]])` - Renvoie `True` si `s` finit par la sous-chaîne *suffix*, `False` sinon. En option *start* et *end* contraigne la recherche. `s.startswith(suffix [,start [, end]])` - `True` si `s` débute par *suffix*.
`s.find(sub [, start [, end]])` - Renvoie le plus petit index de la sous-chaîne sub ou -1 si elle n'est pas présente. En option *start* et *end* limitent la recherche. `s.rfind(sub [, start [, end]])` - Plus haut index.
`s.index(sub [, start [, end]])` - Identique à `find` mais lève une exception `ValueError` si *sub* n'est pas présente. `s.rindex(sub [, start [, end]])` - Identique a `rfind`.
`s.isalnum()` - Renvoie `True` si `s` n'est pas vide et que tous les caractères sont alphanumériques, `False` sinon. `s.isalpha()` - Identique si les caractères sont alphabétiques. `s.isnumeric()` - Identique si les caractères sont numériques. `s.isspace()` - Identique si les caractères sont blancs. `s.islower()` - Identique, si la chaîne est en minuscule. `s.isupper()` - Identique, si la chaîne est en majuscule.
`s.join(iterable)` - Renvoie une chaîne qui est la concaténation des chaînes dans l'objet *iterable*. Le séparateur utilisé est la chaîne `s`.
`s.lower()` - Renvoie une chaîne identique à `s` en minuscule. `s.upper()` - Renvoie une chaîne identique à `s` en majuscule.
`s.lstrip([chars])` - Renvoie une chaîne identique à `s` dans laquelle tous les espaces à gauche sont supprimées. En option *chars* donne la liste des caractères à supprimer. `s.rstrip([chars])` - Espaces de droite. `s.strip([chars])` - Espaces de gauche et de droite.
`s.replace(new,old)` - Renvoie une chaîne identique à `s` où les occurrences de la sous-chaîne *old* sont remplacées par la sous-chaîne *new*.
`s.split(sep)` - Renvoie une liste des sous-chaînes en utilisant *sep* comme séparateur.
`s.zfill(width)` - Renvoie une copie de `s` rempli à gauche avec des '0' pour que la longueur de la chaîne soit `width`. Les signes + et - sont placés avant les '0'.

Format

La méthode `format` s'appelle sur une chaîne. Elle remplace chaque '`{}`' par les paramètres que l'on donne à la fonction. Il est possible de nommer les paramètres pour plus de clarté.

```
'{}, {}, {}'.format('a', 'b', 'c')          #"a, b, c"
'{2}, {1}, {0}, {2}'.format('a', 'b', 'c')  #"c, b, a, c"
'{lat}, {long}'.format(lat='37N',long='111W')  #"37N, 111W"
"{:c}{:c}{:c}".format(97, 98, 99)  #'abc'
"{:.1e}, {:0.3f}".format(100,3.14)   #'1.0e+02, 3.140'
"{n:b}, {n:o}, {n:x}-{n:X}".format(n=26) #'11010, 32, 1a-1A'
```

Ensembles et Dictionnaires

Set et Frozenset

Le type `set` représente les ensembles d'objets uniques. Il est iterable et mutable. On utilise `set()` ou `set(iterable)` pour créer un ensemble vide ou déjà rempli. L'équivalent non-mutable est `frozenset`. La notation par accolades : `{v1, v2,...}` crée un `set`. Attention, `{}` crée un `dict` (voir plus bas).

Opérations sur les `set` et `frozenset`

	Opérations	Descriptions
Pour set et frozenset	`len(s)`	Cardinal de `s`
	`x in s` `x not in s`	**True** si $x \in s$, **False** sinon **True** si $x \notin s$, **False** sinon
	`s == o` `s != o`	**True** si $s \bigcap o = s$, **False** sinon **True** si $s \bigcap o = \emptyset$, **False** sinon. Aussi noté `s.isdisjoint(o)`
	`s <= o`	**True** si $s \subseteq o$, **False** sinon. Aussi noté `s.issubset(o)`
	`s >= o`	**True** si $o \subseteq s$, **False** sinon. Aussi noté `s.issuperset(o)`
	`s < o` `s > o`	**True** si $s \subseteq o$ et $s \neq o$, **False** sinon **True** si $o \subseteq s$ et $s \neq o$, **False** sinon

	`s | o | ...`
	`s & o & ...`
	`s - o - ...`
	`s ^ o`
		Crée le **set** union de `s,o,…`
		Crée le **set** intersection de `s,o,…`
		Crée le **set** avec les éléments de `s` moins ceux de `o,…`
		Crée le **set** avec les éléments uniquement dans `s` ou `o`.
		Aussi notés : `s.union(o,…)` `s.intersection(o,…)` `s.difference(o,…)` `s.symetric_difference(o)`

	`s.copy()`	Renvoie une copy superficielle de `s`.

Pour set uniquement	`s |= o | ...`
	`s &= o & ...`
	`s -= o - ...`
	`s ^= o`
		Semblable aux opérations ci-dessus mais en modifiant `s`.
		`s.update(o,…)`, `s.intersection_update(o,…)`
		`s.difference_update(o,…)`
		`s.symmetric_difference_update(o)`

	`s.add(x)`	Ajoute l'élement `x` à `s`.
	`s.remove(x)` `s.discard(x)`	Supprime l'élément `x` de `s`. L'opération `remove` lève une exception `KeyError` si `x` n'est pas dans `s`.
	`s.pop()`	Retire et renvoie un élément arbitraire de `s`. `KeyError` si s est vide.
	`s.clear()`	Vide `s`.

Dict

Un `dict` est une table d'associations clef-valeur. Associer une valeur à une clef existante remplace l'ancienne valeur associée. Une clef est un objet *hashable* (cela interdit `list` ou `dict`). Un `dict` est *itérable* sur ses clefs. On utilise les accolades pour créer un `dict` vide `{}` ou rempli : `{key:val, …}`.

Opérations sur les `dict`

Opérations	Description
`len(d)`	Nombre d'associations
`d[k]` `d.get(k [,x])`	Renvoie la valeur associée à `k`. S'il n'existe pas d'association avec la clef `k`, `d[k]` lève une exception `KeyError` alors que `get` renvoie `None` ou la une valeur `x` si elle est spécifiée.
`d[k]=v`	Associe `v` à `k`
`k in d` `k not in d`	**True** si `d` a une clef `k`, **False** sinon Équivalent à **not** `k in d`
`iter(d)`	Itérateur sur les clefs de `d`
`d.clear()`	Supprime toute association de `d`
`d.copy()`	Copie superficielle de `d`

`d.keys()`
`d.values()`
`d.items()`
	Revoient des *vues itérables* de `d` avec les clefs, les valeurs ou les couples clef-valeur. Une vue est dynamique, elle reflète les changements qui pourraient survenir sur le dictionnaire.

`d.popitem()`	Supprime et renvoie arbitrairement une association clef/valeur
`d.pop(k [,x])`	Supprime et renvoie la valeur associé à `k` ou si elle n'existe pas renvoie la valeur `x` si elle spécifiée ou lève une exception `KeyError`.

Contrôles de Flux

Blocs de Code et Structures de Contrôle

Un *bloc de code* est un groupement d'au moins une instruction précédées de "`:`" où chaque instruction est *indenté*. L'instruction **pass**, ne réalisant rien, permet de creer des blocs ne faisant rien. Un bloc peut en inclure d'autres.

Structures de contrôle

	Description	Syntaxe

Instruction Conditionnelle	Le bloc **if** est exécuté si *condition* est **True**. Sinon, le premier des blocs optinels **elif** associé à une condition **True** sera exécuté.
	Si toute condition est fausse, le bloc optionel final **else** sera exécuté.
	Pas plus d'un bloc d'instructions n'est exécuté.
		`if condition :`
		`…`
		`elif condition :`
		`…`
		`else :`
		`…`

Boucle Conditionnelle	Le bloc **while** est répété tant que *condition* est **True**.
	Le mot clef **break** entraine une sortie immédiate de la boucle tandis que **continue** passe à l'itération suivante.
	Le bloc **else** est optionel. Il exécuté à la sortie du **while** sauf si c'est avec **break**.
		`while condition :`
		`…`
		`else :`
		`…`

Boucle pour chaque	Le bloc est exécuté pour chaque valeur *i* contenu dans l'objet *iterable*.
	else, **break** et **continue** fonctionnent comme pour la boucle **while**.
		`for i in iterable :`
		`…`
		`else :`
		`…`

Exceptions et Contextes

Les *exceptions* sont un moyen de gérer les instructions susceptibles de ne pas fonctionner correctement. La table suivante décrit la protection d'un bloc de code. Pour *lancer* (ou propager) une exception, on utilise le mot clef **raise** suivit d'une exception : `raise Exception()`.

Protection d'un bloc de code

Description	Syntaxe

Le bloc **try** contient les instructions à protéger.
Au moins un bloc **except** pour attraper des exceptions et exécuter un bloc de code le cas échéant. *Exc* est le type de l'exception détectée. Au besoin, elle est stockée dans une variable (**as** e).
Le bloc **else** est exécuté en cas de sortie normal.
Le bloc **finally** est exécuté dans tous les cas.
	`try:`
	`…`
	`except Exc:`
	`…`
	`except Exc as e:`
	`…`
	`else:`
	`…`
	`finally:`
	`…`

Un *contexte* est un bloc de code protégé associé à un objet : le *context manager*. Cet objet possède les méthodes `__enter__(self)` et `__exit(self)__`. À l'entrée du contexte, `__enter__` est appelée. Si cette dernière s'exécute sans erreur, `__exit__` sera appelée à la sortie du bloc quelque soit son déroulement.

Pour définir un contexte on utilise **with** suivit d'une expression renvoyant le context manager. S'il doit être utilisé dans le bloc de code, on peut le lier à une variable avec **as**.

```
with expr [as e] :
    … # instruction pouvant faire référence à e
```

Voir aussi la colonne *Entrée/Sortie* pour un exemple d'utilisation.

Fonctions

Définition

Une fonction est un bloc d'instructions (*corps d'une fonction*) destinés à réaliser une opération. Elle peut recevoir des *paramètres* et renvoie une valeur. Une fonction est un objet et peut être manipulé comme toute autre valeur.

```
def fctName(pParam, …, *args, nParam=val, …, **kwargs):
    instructions
    …
    return values
```

Une fonction a toujours une *valeur de retour*, par défaut **None**. C'est la valeur qui sera substitué à l'appel de la fonction. Pour spécifier la valeur de retour on utilise le mot clef **return** suivit d'une ou plusieurs valeurs regroupées dans un `tuple`.

Paramètres

Il peut y avoir un nombre fixe, éventuellement aucun, de *paramètres positionnels obligatoires* (tel que *pParam*). Ce sont toujours les premiers d'une fonction et doivent être présents dans le bon ordre lors de l'appel d'une fonction.

Un nombre variable de *paramètres positionnels optionnels* peut être demandé avec '*' et conventionnellement le nom de variable *args*. À l'appel de la fonction on pourra ajouter un nombre quelconque de paramètres positionnels à la suite des obligatoires. Ils seront stockés dans le `tuple` *args*.

Suivent les *paramètres nommés* avec valeurs par défaut tel que *nParam*. Ces paramètres sont facultatifs. Lorsqu'ils ne sont pas présents à l'appel, Python leur assignera la valeur par défaut indiquée dans la définition de la fonction.

Les *paramètres nommés optionnels* sont indiqués avec '**' et conventiellement le nom de variable *kwargs*. À l'appel, tout paramètre nommé ne correspondant pas à un paramètre avec une valeur par défaut sera ajouté au `dict` *kwagrs*.

Appel

Pour appeler une fonction, on fait suivre son nom de parenthèses contenant la liste des paramètres. Les parenthèses sont obligatoires même s'il n'y a aucun paramètre. L'ordre est important : paramètres positionnels obligatoires, paramètres positionnels optionnels et enfin paramètres nommés.

```
fctName(pParam, …, *args, nParam=val, **kwargs )
```

Lorsque l'on utilise une fonction sans la faire suivre de parenthèses on désigne la fonction elle même plutôt que son appel.

Lambda, Fonction Anonyme

Une fonction est un objet comme un autre. On peut donc la passer en paramètre d'une autre. On accède à une fonction à partir de son nom qui est une variable. Le mot clef lambda permet de créer des fonctions anonymes, c'est-à-dire qui ne sont pas lié à une variable. La syntaxe pour un ou plusieurs paramètres est :

```
lambda x : expr(x)
lambda x, y, … : expr(x,y,…)
```

Modules

Modules, Packages et Import

Un *module* est un fichier *nom.py* qui regroupe définitions de constantes, fonctions et classes. Un *package* est un ensemble de modules réunis dans un dossier contenant un fichier `__init__.py`. On importe modules et packages à partir de leur *nom* avec **import**. Les packages sont recherchés dans le `PYTHONPATH` qui contient le répertoire courant accessible dans la variable `sys.path`.

*Différentes utilisation pour **import***

Instructions	Descriptions
`import mod`	Importe un module. Notation préfixée pour accéder aux fonctions, classes, .. : *mod.fonction*.

`from mod import f`
`from mod import f, g`
`from mod import *`
`__all__ = [f, g, ...]`
	Importe seulement les éléments indiqués. Utilisation sans préfixer par le nom du module. Un module entier est importé avec *, sauf si son effet est réduit par la définition de la constante `__all__`.

`from mod import fct as f`	Comme ci-dessus mais *fct* est renommé *f*.

Modules Utiles

	Fonctions	Description
math	`log(x)  log(x,y)`	Logarithme népérien (ou en base *y*) de *x*.
	`sqrt(x)`	Racine carré de *x*.
	`hypot(x,y)`	Hypoténuse de *x* et *y*.
	`floor(x)  ceil(x)`	Partie entière inférieure et supérieure de *x*.
	`factorial(x)`	Factorielle de *x*.
	`sin(x)  cos(x)` `tan(x)`	Fonctions trigonométriques usuelles, en radian. Également :`asin(x)`, `sinh(x)`,...
	`degrees(x)`	Conversion de *x*, de radians vers degrés
	`randian(x)`	Conversion de *x*, de degrés vers radians
random	`seed(x)`	Utilise *x* pour initialiser la graine du générateur.
	`random()`	Génère un `float` suivant une distribution unif. sur [0,1[.
	`randint(a,b)`	Génère un `int` suivant une distribution unif. sur [*a*,*b*].
	`uniform(a,b)`	Génère `float` suivant une distribution uniforme sur [*a*,*b*[.
	`choice(seq)`	Sélectionne un élément parmi une séquence non vide.
	`shuffle(l)`	Mélange les éléments de `l`. Ne renvoie rien.
	`sample(pop,x)`	Renvoie une liste de *x* éléments tirés dans *pop*.
os	`chdir(path)`	Change le répertoire courant qui devient `path` (**str**).
	`getcwd()`	Renvoie le répertoire courant (**str**).
	`listdir(path)`	Liste les fichiers contenus dans le répertoire *path*.
	`mkdir(path)`	Crée un répertoire *path*.
	`remove(path)`	Supprime le fichier *path*.
	`rename(src,dst)`	Déplace le fichier *src* vers *dst*.
	`rmdir(path)`	Supprime le répertoire *path*.

io	`f=open("nom"`
	`[,mode]`
	`[,encoding=..])`
		Renvoie un `file` correspondant au fichier *nom* ouvert en lecture. On peut changer le mode : "r" (lecture), "w" (écriture), "+" (lecture/écriture), "a" (ajout), "b" (binaire).

	`f.write(s)`	Écrit la chaîne s dans le fichier f.
	`f.read([n])`	Renvoie le contenu de *f* ; *n* pour limiter les données lues.
	`f.readline()`	Renvoie une ligne du fichier.
	`f.readlines()`	Renvoie une liste des lignes du fichier.
	`f.flush()`	Vide le buffer d'écriture.
	`f.close()`	Vide le buffer d'écriture et ferme le fichier.
sys	`argv`	Liste des arguments du programme.
	`exit([s])`	Quitte le programme. La valeur de sortie est *s* sinon `0`.
	`path`	Liste de chaîne de caractères contenant le `PYTHONPATH`.
	`stdin stdout sterr`	Entrée/sortie standard et d'erreur. De type `file`.

Objets et Classes

Structure Générale

En python *tout est objet*, un entier, une fonction ou encore un module. Une *classe* est un moule permettant de créer des *objets* partageant des propriétés. Elle est définie avec **class** suivit de son nom (par convention *CamelCase*), d'éventuelles classes mères (voir *héritage*) et un bloc de code.

```
class ClassName( [ParentClasses] ):
    #bloc de la classe détaillant des propriétées communes
```

On peut ajouter des *attributs* à un objet *o* (sauf s'il est de type *builtin*). Ce sont des variables accessibles lié à l'objet *o*. On y accède avec la notation "point" : *o.attr = expr* pour donner une valeur, *o.attr* pour y accéder dans une expression ou **del** *o.attr* pour le détruire. Le plus courant est que chaque objet d'une classe dispose des mêmes attributs. Pour cela on les définit dans une méthode spéciale `__init__`.

```
def __init__(self [, params]) :   #self désigne l'objet en
    self.var_i = expr             #cours de création
```

Pour construire un objet de la classe *ClassName*, on utilise l'instruction suivante : *ClassName*([*params*]). Elle aura pour conséquence l'allocation de l'espace nécessaire à l'objet et son initialisation par appel implicite à la méthode `__init__` avec comme paramètre l'objet en cours de création et les paramètres *params*.

Une classe est elle même un objet. Elle peut donc disposer d'attributs, accessible par la notation "point" : *ClassName.attr_classe*. Tout objet *o* de type *ClassName* peut accéder aux attributs de *ClassName* : *o.class_attr*. Attention, on ne distingue pas explicitement l'utilisation des attributs liés à la classe de ceux liés à l'objet.

Fonctions

Tout objet (et donc également toute classe) dispose d'attributs de type fonction, on parle de *méthode*. Pour appeler une méthode, on utilise la notation point. On distingue deux cas. Lorsque l'on appelle une méthode sur un objet, *o.method(params)*, l'objet sera passé implicitement comme premier paramètre positionnel. Lorsque l'on appelle une méthode sur un objet de type classe, *ClassName.static_method(params)*, aucun paramètre implicite n'est ajouté en paramètre. On parle de *méthode statique*.

En plus d'un certain nombre de méthodes déjà existantes, il est possible d'associer à un objets de nouvelles méthodes. On peut le faire indépendamment pour chaque objet avec la notation "point" vu plus haut en ajoutant un attribut de type fonction. Dans la conception objet, on tend à ce que tous les objets d'une certaine classe partage les mêmes méthodes. Elles sont donc définies de manière commune dans le bloc de définition de la classe.

Héritage

L'héritage permet de donner à une *classe fille* un ensemble de propriétés issues de *classes mères*. Ainsi, une classe fille hérite des méthodes de ses classes mères. Lorsqu'un objet d'une de type classe fille, appelle une méthode, celle-ci va d'abord être recherchée dans la classe fille puis dans les classes mères par ordre d'apparition dans la déclaration de la classe fille.

Le fonctionnement des méthodes issues des classes mères nécessite souvent l'ajout d'un certain nombre d'attributs présents dans les classes mères à la classe fille. Cela peut être fait dans la méthode `__init__`, en appelant les méthodes `__init__` des classes mères : *ParentClass.*`__init__`(**self**,...). Attention, ces méthodes sont appelées dans l'ordre choisi par le développeur, ainsi certain attributs peuvent être initialiser plusieurs fois et donc "écrasé. Il faut donc être vigilent en cas d'héritage multiple.

Objets et Classes (Suite)

Exemples de Classe

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

    def __str__(self) :
        return str( "({},{})".format(self.x, self.y) )

class Rectangle(object):
    def __init__(self, coin, lar, hau):
        self.coin = coin
        self.lar = lar
        self.hau = hau

    def trouveCentre(self):
        cx = self.an.x + self.lar / 2
        cy = self.an.y + self.hau / 2
        return Point(xc,yc)

class Carre(Rectangle):
    def __init__(self, coin, cote):
        Rectangle.__init__(self, coin, cote, cote)

    def surface(self):
        return self.lar**2

c = Carre( Point(0,0), 10 )
print( "Centre du carré", c.trouveCentre() )
```

Méthodes Spéciales

Les méthodes spéciales permettent de spécifier le comportement des objets par rapport à certains opérateurs ou fonction du langage. Elles commencent et finissent toutes par deux caractères soulignés :'__'.

Remarque : lors de l'utilisation d'un opérateur binaire (ex. *x+y*), la méthode `__add__` est appelée sur *x*, si celle ci est non définie alors c'est `__radd__` qui est appelée sur *y*. Dans les deux cas un nouvel objet est crée. Pour l'affectation raccourcie *x+=y*, la méthode appelée est `__iadd__`. Elle modifie la valeur de *x*.

Liste de méthodes spéciales usuelles.

```
  x<y → __lt__(self,y)
 x=<y → __le__(self,y)
 x==y → __eq__(self,y)
 x!=y → __ne__(self,y)
  x>y → __gt__(self,y)
 x=>y → __ge__(self,y)

  x+y → __add__(self,y)
  x-y → __sub__(self,y)
  x*y → __mul__(self,y)
  x/y → __truediv__(self,y)
 x//y → __floordiv__(self,y)
  x%y → __mod__(self,y)
 x**y → __pow__(self,y)
 x<<y → __lshift__(self,y)
 x>>y → __rshift__(self,y)
  x&y → __and__(self,y)
  x|y → __or__(self,y)
  x^y → __xor__(self,y)

   -x → __neg__(self)
   +x → __pos__(self)
abs(x)→ __abs__(self)
   ~x → __invert__(self)
```

```
      str(o) → __str__(self)
     repr(o) → __repr__(self)
format(o, s) → __format__(self, s)
     bool(o) → __bool__(self)
      int(o) → __int__(self)
    float(o) → __float__(self)
  complex(o) → __complex__(self)
  round(o,n) → __round__(self,n)

        o.n → __getattr__(self,n)
      o.n=v → __setattr__(self,n,v)
    del o.n → __delattr__(self,n)
    hash(o) → __hash__(self)

      len(o) → __len__(self)
       o[k] → __getitem__(self,k)
     o[k]=v → __setitem__(self,k,v)
   del o[k] → __delitem__(self,k)
reversed(o) → __reversed__(self)
     x in o → __contains__(self,x)
for i in o: → __iter__(self)
    next(g) → __next__(self)

       with → __enter__(self)
            → __exit__(self)
```

Opérations Avancées

Range

La classe `range` permet de construire des itérables pour parcourir des valeurs entières. Son constructeur prend jusqu'à trois paramètres : `range(`*fin*`)`, `range(`*deb*`,`*fin*`)` ou `range(`*deb*`,`*fin*`,`*pas*`)`. Ex. : "`for i in range(10)`".

Attention, `range` n'est pas un générateur (voir ci-dessous).

Générateurs

Un *générateur* est une description d'une suite de valeurs qui ne seront générées qu'à leur utilisation, un générateur consomme donc peu de mémoire.

```
( expr(var) for var in iter )
( expr(var) for var in iter if condition(var)  )
```

On égraine les valeurs d'un générateur en appelant la fonction `next(`*generator*`)` qui renvoie un nouvelle valeur de *generator* ou en l'utilisant comme itérable dans une boucle : `for` *i* `in` *generator*.

On peut également utiliser le mot clef **yield** pour créer des générateurs. Le fonctionnement est semblable a la déclaration d'une fonction mais on utilise **yield** à la place de **return**. La fonction est mise en pause à chaque instruction **yield** et l'exécution reprend du même endroit à chaque appel de `next`.

```
def fib(max) :
    a,b = 0,1
    while a < max :
        yield a
        a,b = b, a+b
```

```
for i in fib(100):
    print(i)
```

On peut forcer la construction de valeurs en transformant le générateur en `list` ou en `tuple` avec les méthodes de conversion (`list(g)`, `tuple(g)`). On peut également utiliser les notations suivantes pour construire directement des listes et dictionnaires. On parle de liste (et dictionnaire) *en compréhension*.

```
tuple(i for i in range(10) if i%2==0)  #(0,2,4,6,8)
[[j for j in range(4)] for i in range(4)]  #matrice 4x4
{i:i*i for i in range(5)}  #{0:0, 1:1, 2:4, 3:9, 4:16}
```

Opérations sur des Objets Itérables

Fonctions sur les itérables

Fonctions	Descriptions
`all(`*iter*`)` `any(`*iter*`)`	**True** si tout élément de *iter* est **True**, sinon **False**. **True** si au moins un élément est **True**, sinon **False**.
`enumerate(`*iter*`)`	Construit un `iterator` contenant les tuples index,objet de tous les objets de *iter*.
`filter(`*function*`,`*iter*`)`	Construit un `iterator` avec les éléments contenus dans *iter* pour lesquels *function* renvoie **True**.
`map(`*function*`,`*iter*`)`	Construit un `iterator` qui applique *function* à tous les objets de *iter*.
`sorted(`*iter*`[,`*reverse*`])`	Construit une `list` triée par ordre croissant ou décroissant si *reverse* est **True**.
`zip(*`*iterables*`)`	Construit un `iterator` qui aggrège tous les éléments des itérables en argument.

La fonction **enumerate** est particulièrement utile pour le traitement d'un itérable pour lesquel il est nécessaire de manipuler à la fois l'indice et la valeur des éléments. En effet, une boucle **for** peut porter sur plusieurs itérables de même longueur ou un itérable contenant des tuples. On utilise plusieurs variables de contrôle.

```
for index, value in enumerate(iterable) :
    …
```

Autres

Entrées/Sorties Standard

La fonction `print(*`*objects*`)` permet d'écrire des chaînes de caractères sur la *sortie standard* d'un programme : la console. Cette fonction appelle la méthode `__str__` de chaque paramètre pour le convertir en chaîne avant de l'afficher.

La fonction `input(`[*prompt*]`)` permet d'écrire sur l'*entrée standard* : le clavier. Cette fonction renvoie les caractères saisis jusqu'au premier retour à la ligne. En option elle prend une chaîne *prompt* qui sera affichée avant.

Lecture/Écriture dans des Fichiers

Les fonctions utilisées pour manipuler des fichiers sont contenues dans le package `io`. Ce sont `open`, `close`, `read`, `readline`, `readlines` et `write`. Ces fonctions sont accessibles sans importer explicitement le module `io`. Voir dans la colonne *Module*, la section *Module Utile* pour les détails de ces fonctions.

La lecture/écriture s'utilise de préférence avec un manageur de contexte (voir colonne *Contrôle de Flux*. Dans l'exemple, suivant la fonction `close` est appelée implicitement à la sortie du contexte.

Un objet fichier est *itérable* pouvant être utiliser pour lire les lignes une à une.

```
with open("/chemin/du/fichier.ext") as f :
    print( f.readline() )  #lit la première ligne de f
    for l in f :           #lit toutes les suivantes
        print( l )
```

Conversions de Type

Python dispose de fonctions pour les conversions entre type. Le comportement des ces fonctions peut être défini pour chaque objet grace aux méthodes spéciales correspondantes (voir colonne *Objets et Classes (Suite)*).

Les fonctions de conversion vers les types numériques sont `int(o)`, `float(o)` et `complex(o)`. Pour la conversion en chaîne de caractères, deux fonctions sont disponibles : `str(o)` et `repr(o)`. La première donne une représentation lisible, la seconde une représentation sans ambiguité. La fonction `str` est implicitement appelé lors de l'appel de `print(o)` pour convertir *o* en chaîne.

La fonction `ord(c)` renvoie l'entier correspondant au code unicode associé à un caractère. L'opération inverse est `chr(i)`.

Expression Conditionnelle

Une expression conditionnelle a une valeur dépendant d'une condition. Sa syntaxe est la suivante : *true_val* **if** *condition* **else** *false_val*

Si *condition* est vraie, cette expression vaudra *true_val*, sinon elle faudra *false_val*. Voici deux exemples équivalents.

```
abs = x if x>=0 else -x
```

```
if x>=0 :
    abs = x
else :
    abs = -x
```

À Propos

Ce memento est un outil d'aide à la programmation. Il tente de regrouper la syntaxe des principaux concepts de Python. Il s'impire beaucoup de "l'Abrégé Dense Python 3.2" de *Laurent Pointal* et plus partiellement du livre "Apprendre à Programmer avec Python" de *Gérard Swinnen*.

Merci à tous les relecteurs.