Une version simplifiée pour le lycée par L. Leskow
du Mémento v2.0.7 de Laurent Pointal

Mémento Python 3

Version originale sous licence CC4
https://perso.limsi.fr/pointal/python:memento

Types de base

entier, flottant, booléen, chaîne

```
int   783   0  -192   0b010   0xF3
                zéro   binaire  hexa
float 9.23  0.0  -1.7e-6
                      ×10^-6
bool  True  False
str   "Un\nDeux"
            retour à la ligne
```

immutables

Conversions

`type(`*expression*`)`

```
int("15") → 15
str(12) → "13"
int(15.56) → 15        troncature de la partie décimale
list("abc") → ["a","b","c"]
':'.join(["toto","12","pswd"]) → "toto:12:pswd"
chr(64)→"@"  ord("@")→64        code ↔ caractère
```

Identificateurs

pour noms de variables, fonctions, modules, classes…

`a…zA…Z_` suivi de `a…zA…Z_0…9`

- accents possibles mais à éviter
- mots clés du langage interdits
- distinction casse min/MAJ

☺ `a toto x7 y_max BigOne`
☹ ~~`0y and for`~~

Aide

```
help(help)
dir(truc)
```

affiche aussi votre propre documentation

Types conteneurs

- **séquences ordonnées**, accès par index

```
tuple (1,5,9)   Valeurs non modifiables (immutables)
list  [1,5,9]   Valeurs modifiables (mutables)      []  liste vide
```

liste en compréhension :

```
[i ** 2 for i in range(5)] → [0,1,4,9,16]
[i for i in range(7) if i % 2 == 0] → [0,2,4,6]
```

- **conteneurs clés**, sans ordre *a priori*, accès par clé rapide, chaque clé unique

```
dictionnaire  dict {"clé":"valeur"}   Mutables      {}  dictionnaire vide
```

Variables & affectation

`=`

affectation ⇔ **association** d'un *nom* à une *valeur*
1) évaluation de la valeur de l'expression de droite
2) affectation dans l'ordre avec les noms de gauche

```
x = 1.2 + 8 * y
y, z, r = 9.2,-7,0      affectations multiples
a, b = b, a             échange de valeurs
```

Indexation conteneurs séquences

pour les listes, chaînes de caractères, tuples …

index positif	0	1	2	3	4
`lst=`	`[10,`	`20,`	`30,`	`40,`	`50]`

Sur les séquences modifiables (`list`) modification par affectation `lst[4]=25`

Accès individuel aux **éléments** par `lst[`*index*`]`

`lst[0]`→`10`
`lst[1]`→`20`

Nombre d'éléments `len(lst)`→`5`

`lst.append(`*val*`)` ajout d'un élément à la fin

Séquences d'entiers

`range(`[*début*,] *fin* [,*pas*]`)`

début défaut 0, *fin* non compris dans la séquence, *pas* signé et défaut 1

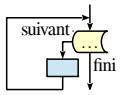
```
range(5) → 0 1 2 3 4        range(2,12,3) → 2 5 8 11
range(3,8) → 3 4 5 6 7      range(20,5,-5) → 20 15 10
```

range fournit une séquence immutable d'entiers construits au besoin

Instruction boucle itérative

bloc d'instructions exécuté ***pour chaque*** *élément d'un conteneur ou d'un itérateur*

```
for var in séquence:
    bloc d'instructions
```



Parcours des **valeurs** d'un conteneur

```
s = "Du texte"      } initialisations avant la boucle
cpt = 0
```

variable de boucle, ***affectation gérée par l'instruction `for`***

```
for c in s:
    if c == "e":
        cpt = cpt + 1
print("trouvé",cpt,"'e'")
```

Algo : comptage du nombre de e dans la chaîne.

bonne habitude : ne pas modifier la variable de boucle

Maths

nombres flottants… valeurs approchées !

Opérateurs : `+ - * / // % **`
Priorités `(…)` : × ÷ (↑) ÷ entière, reste ÷ (↑) a^b

```
(1+5.3)*2→12.6   abs(-3.2)→3.2
from math import sin,pi…
sin(pi/4)→0.707…
cos(2*pi/3)→-0.4999…
sqrt(81)→9.0
```

Op. sur dictionnaires

```
d[clé]=valeur
d[clé]→ valeur
```

vues itérables sur

```
d.keys()    } les clés
d.values()  } valeurs
d.items()   } couples
```

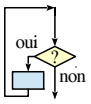
`d.get(`*clé*[,*défaut*]`)` → *valeur*

Instruction boucle conditionnelle

attention aux boucles sans fin !

bloc d'instructions exécuté ***tant que*** *la condition est vraie*

```
while condition logique:
    bloc d'instructions
```




```
s = 0   } initialisations avant la boucle
i = 1   } condition avec au moins une valeur variable (ici i)
while i <= 100:
    s = s + i**2
    i = i + 1       faire varier la variable de condition !
print("somme:",s)
```

Algo : $s = \sum_{i=1}^{i=100} i^2$

Imports modules/noms

module `truc` ⇔ *fichier* `truc.py`

```
from monmod import nom1,nom2 as fct
```

→*accès direct aux noms, renommage avec* `as`

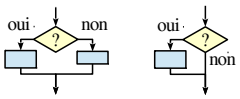
```
import monmod   →accès via monmod.nom1 …
```

modules et packages cherchés dans le *python path* (cf. `sys.path`)

Instruction conditionnelle

un bloc d'instructions exécuté, uniquement ***si*** *sa condition est vraie*

```
if condition logique:
    bloc d'instructions
```



Combinable avec des *sinon si, sinon si…* et un seul *sinon* final. Seul le bloc de la première condition trouvée vraie est exécuté.

avec une variable `x`*:*

```
if bool(x)==True:   ⇔ if x:
if bool(x)==False:  ⇔ if not x:
```

```
if age<=18:
    etat="Enfant"
elif age>65:
    etat="Retraité"
else:
    etat="Actif"
```

Logique booléenne

Comparateurs: `< > <= >= == !=` (*résultats booléens*) ≤ ≥ = ≠

`a and b` et logique — *les deux en même temps*
`a or b` ou logique — *l'un ou l'autre ou les deux*

piège : `and` *et* `or` *retournent la valeur de* `a` *ou de* `b` *(selon l'évaluation au plus court).*
⇒ *s'assurer que* `a` *et* `b` *sont booléens.*

`not a` non logique

`True` `False` constantes Vrai/Faux

Blocs d'instructions

instruction parente :
bloc d'instructions 1…
instruction parente :
bloc d'instructions 2…
instruction suivante après bloc 1

indentation !

régler l'éditeur pour insérer 4 espaces à la place d'une tabulation d'indentation.

Définition de fonction

nom de la fonction (identificateur) — paramètres nommés

```
def fct(x,y,z):
    """documentation"""
    # bloc instructions, calcul de res, etc.
    return res
```

`fct`

`return res` ← valeur résultat de l'appel, si rien à retourner `return None`

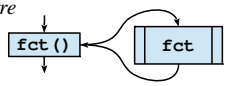
les paramètres et toutes les variables de ce bloc n'existent que *dans* le bloc et *pendant* l'appel à la fonction (penser "*boîte noire*")

Appel de fonction

```
r = fct(3,"hey",a)
```

stockage/utilisation de la valeur de retour — *une valeur d'argument par paramètre*

c'est l'utilisation du nom de la fonction *avec les parenthèses* qui fait l'appel



Affichage

```
print("v=",3,"cm :",x,",",y+4)
```

éléments à afficher : valeurs littérales, variables, expressions
Options de `print`:

- `sep=" "` séparateur d'éléments, défaut espace
- `end="\n"` fin d'affichage, défaut fin de ligne