Introduction à l'algorithmique et à la programmation

IUT 1ère année 2013-2014

Cyril Nicaud

Cyril.Nicaud@univ-mlv.fr

– Cours 1 / 5 –

– Déroulement du cours –

Organisation :

- 5 séances de 2h de cours
- 10 séances de 2h de TD
- 15 séances de 2h de TP

Evaluation :

- une note de TD : il y a une rapide interrogation écrite au début de chaque TD (sauf le premier)
- une note de TP : vous rendez chaque TP, certains seront notés
- un projet de programmation
- un examen

Lorsque vous voyez le python à gauche, cela signifie qu'il y a une démonstration d'écriture et d'exécution de programme pendant le cours. Ces démonstrations ne sont pas sur les transparents.

– Définition dans le Trésor –

Programmer : (Empl. intrans.) “Écrire un programme d’ordinateur” (Ging.-Lauret 1982), fractionner un problème en instructions codifiées acceptables par la machine.

– Introduction sur un exemple –

Un exemple de programme en Python 3 :

```
def puissance(x,n):
    res = 1
    i = 1
    while i <= n:
        res = res * x
        i = i + 1
    return res

y = puissance(12,6)
print(y)
```

C'est un programme un peu avancé ...

– Introduction sur un exemple –

Un exemple de programme en Python 3 :

```
def puissance(x,n):
    res = 1
    i = 1
    while i <= n:
        res = res * x
        i = i + 1
    return res

y = puissance(12,6)
print(y)
```

C'est un programme un peu avancé . . . mais essayons de comprendre ce qu'il fait.

– Les différents mots –

```
def puissance(x,n):
    res = 1
    i = 1
    while i <= n:
        res = res * x
        i = i + 1
    return res

y = puissance(12,6)
print(y)
```

Les couleurs sont un ajout pratique pour la lisibilité, elle peuvent changer d'un éditeur à l'autre. Ici on a :

- en orange des mots-clés du langage Python
- en violet des fonctions du langage Python
- en bleu des noms choisis par le programmeur

– Les différents mots –

```
def puissance(x,n):
    res = 1
    i = 1
    while i <= n:
        res = res * x
        i = i + 1
    return res

y = puissance(12,6)
print(y)
```

Les mots de Python, en orange et en violet, sont en anglais.

def = define = définir	**while** = tant que
return = retourner	**print** = imprimer

– L'indentation –

```
def puissance(x,n):
    res = 1
    i = 1
    while i <= n:
        res = res * x
        i = i + 1
    return res

y = puissance(12,6)
print(y)
```

- Les lignes ne commencent pas toutes au même endroit.
- Ce n'est pas un détail !
- Le positionnement (on dit l'indentation) des lignes est important pour leur signification.

Si on masque temporairement les lignes indentées, on obtient :

```
def puissance(x,n):
    ...
y = puissance(12,6)
print(y)
```

```
def puissance(x,n):
    ...
y = puissance(12,6)
print(y)
```

- **def** définit une fonction qui s'appelle ici puissance
- La fonction puissance(x,n) calcule la valeur x^n (on verra comment plus tard)
- puissance(12,6) appelle la fonction puissance avec les paramètres 12 et 6, qui calcule donc 12^6
- y = puissance(12,6) stocke la valeur 12^6 dans y
- **print** est une fonction de Python qui affiche la valeur de son paramètre à l'écran

Essayons un peu le programme et des variantes ...

– La fonction puissance –

```
def puissance(x,n):
    res = 1
    i = 1
    while i <= n:
        res = res * x
        i = i + 1
    return res
```

L'indentation signifie que toutes les lignes font partie de la fonction puissance

- La fonction prend deux paramètres x et n
- Au début on met la valeur 1 dans res et i
- **while** (= tant que) possède
  - une condition $i \leq n$
  - deux lignes indentées
  - ces deux lignes sont répétées "tant que" la condition est vraie
- **return** (= retourner) termine la fonction en renvoyant la valeur de res

– Détail de la boucle while –

```
while i <= n:
    res = res * x
    i = i + 1
```

Pour l'exemple x = 2 et n = 4

Au début res et i valent 1

– Détail de la boucle while –

```
while i <= n:
    res = res * x
    i = i + 1
```

Pour l'exemple x = 2 et n = 4

Au début res et i valent 1

	x	n	res	i
Avant de commencer	2	4	1	1

– Détail de la boucle while –

```
while i <= n:
    res = res * x
    i = i + 1
```

Pour l'exemple x = 2 et n = 4

Au début res et i valent 1

	x	n	res	i
Avant de commencer	2	4	1	1
Etape 1	2	4	2	2

– Détail de la boucle while –

```
while i <= n:
    res = res * x
    i = i + 1
```

Pour l'exemple x = 2 et n = 4

Au début res et i valent 1

	x	n	res	i
Avant de commencer	2	4	1	1
Etape 1	2	4	2	2
Etape 2	2	4	4	3

– Détail de la boucle while –

```
while i <= n:
    res = res * x
    i = i + 1
```

Pour l'exemple x = 2 et n = 4

Au début res et i valent 1

	x	n	res	i
Avant de commencer	2	4	1	1
Etape 1	2	4	2	2
Etape 2	2	4	4	3
Etape 3	2	4	8	4

– Détail de la boucle while –

```
while i <= n:
    res = res * x
    i = i + 1
```

Pour l'exemple x = 2 et n = 4

Au début res et i valent 1

	x	n	res	i
Avant de commencer	2	4	1	1
Etape 1	2	4	2	2
Etape 2	2	4	4	3
Etape 3	2	4	8	4
Etape 4	2	4	16	5

A la fin, "**return** res" retourne donc la valeur 16 qui est bien 2^4

– Détail de la boucle while –

```
while i <= n:
    res = res * x
    i = i + 1
```

Pour l'exemple x = 2 et n = 4

Au début res et i valent 1

	x	n	res	i
Avant de commencer	2	4	1	1
Etape 1	2	4	2	2
Etape 2	2	4	4	3
Etape 3	2	4	8	4
Etape 4	2	4	16	5

A la fin, "**return** res" retourne donc la valeur 16 qui est bien 2^4

Essayons diverses variantes ...

– Robustesse du programme –

Qu’est-ce qui se passe si n = 0 ? ou x = 4.5 ? ou n = -4 ?

Essayons ...

– Robustesse du programme –

Qu'est-ce qui se passe si n = 0 ? ou x = 4.5 ? ou n = -4 ?

Essayons ...

On va corriger le problème si n est négatif, en rajoutant un test et en traitant le cas séparément. On va ajouter au début de puissance :

```
if n < 0:
    print('erreur : pas de n négatif')
    return
```

- if (= si) est un test, ici on regarde si n est strictement négatif
- les deux lignes indentées ne sont effectuées que si la condition du if est vraie
- si n < 0, on affiche un message d'erreur et return termine la fonction en ne retournant rien

– La nouvelle fonction puissance –

```
def puissance(x,n):      # calcule x puissance n
    if n < 0:            # cas n < 0 non géré
        print('erreur : pas de n négatif')
        return
    res = 1
    i = 1
    while i <= n:        # on fait n fois
        res = res * x    # multiplier res par x
        i = i + 1        # ajouter 1 à i
    return res
```

- On a ajouté des commentaires :
  - le caractère # indique que la suite de la ligne est un commentaire
  - les commentaires sont ignorés par Python
  - ils servent à décrire le programme pour les êtres humains qui le lisent
- Les commentaires sont très importants en programmation ... on y reviendra

– En résumé –

- On a vu un exemple avancé, qui nous a permis de voir, dans les grandes lignes, à quoi ressemble un programme
- On a vu
  - que les lignes sont des instructions à effectuer, dans l'ordre
  - qu'on peut stocker des valeurs (dans y, x, ...)
  - qu'on peut effectuer des calculs (res * x)
  - que des lignes peuvent être effectuées plusieurs fois (**while**), ou seulement si une condition est vérifiée (**if**)
  - qu'on peut créer des fonctions (puissance) ou utiliser des fonctions de Python (**print**)
  - ...

– En résumé –

- On a vu un exemple avancé, qui nous a permis de voir, dans les grandes lignes, à quoi ressemble un programme
- On a vu
  - que les lignes sont des instructions à effectuer, dans l'ordre
  - qu'on peut stocker des valeurs (dans y, x, ...)
  - qu'on peut effectuer des calculs (res * x)
  - que des lignes peuvent être effectuées plusieurs fois (**while**), ou seulement si une condition est vérifiée (**if**)
  - qu'on peut créer des fonctions (puissance) ou utiliser des fonctions de Python (**print**)
  - ...

- Dans la suite, on va apprendre pas à pas et dans le détail toutes ces notions (et bien d'autres), afin que vous maîtrisiez les bases de la programmation

Python pas à pas

Variables, types et opérations

– Types de valeurs –

- Les valeurs de base possèdent un type
- Le type va notamment déterminer ce qui se passe quand on fait une opération sur des valeurs

Les principaux types :

- entier (**int**) : **12** **-4** **123545** ...
- flottant (**float**) : **3.14159** **-1.5** **12.** **4.56e12** ...
- booléen (**bool**) : **True** (vrai) ou **False** (faux)
- indéfini, rien : **None**
- chaîne de caractères (**str** pour "string") : **'chaîne de caractères'** **'IUT info'**, ...

Les majuscules/minuscules sont importantes :
True $\neq$ **true**

– Transtypage –

- La fonction **type**() permet de connaître le type d'une valeur
- On peut demander à Python de changer le type d'une valeur
- On peut par exemple toujours transformer une valeur de base en chaîne de caractères avec la fonction **str**()
- Par exemple **str**(51) renvoie la chaîne **'51'**
- Attention : le nombre 51 et la chaîne **'51'** ce n'est pas la même chose pour Python. On y reviendra.
- **int**() convertit en entier, quand cela est possible
- **float**() convertit en flottant, quand cela est possible
- **bool**() convertit en booléen

Essayons dans un terminal Python ...

– Quelques exemples –

int (4.5) → 4	**int** (-4.5) → 4	**int** (**'0345'**) → 345
int (**'IUT'**) → **erreur**	**float** (4) → 4.	**float** (**'4.5'**) → 4.5
str (4) → **'4'**	**str** (**True**) → **'True'**	**str** (-4.5) → **'-4.5'**
bool (4) → **True**	**bool** (0) → **False**	**bool** (**'IUT'**) → **True**

En pratique, on se sert surtout de :

- **str** qui fonctionne tout le temps
- **int** et **float** appliqués à une chaîne de caractères qui correspond à un nombre
- **int** appliqué à un **float** pour tronquer les décimales

– Opérations sur les nombres –

- Sur les **int** et sur les **float** on a l'addition **+**, la soustraction **-**, la multiplication ***** et la division **/**
- Si on compose deux **int** on obtient un **int** , sauf la division qui renvoie un **float**
- Si on compose deux **float** , ou un **int** et un **float** , on obtient un **float**
- On dipose également de la division Euclidienne, avec quotient et reste comme en primaire. Le quotient de x et y est **x // y** et leur reste est **x % y**
- Il y a enfin l'opération puissance qui se note **x ** y**
- Les opérations suivent les règles de priorités usuelles et on peut utiliser des parenthèses : **(4+2)*1.2**

Quelques exemples ...

– Opérations avec booléens –

- On a les opérations sur les booléens :
  - **and** c'est le **ET** logique, **x** **and** **y** vaut **True** seulement quand **x** et **y** valent **True**
  - **or** c'est le **OU** logique, **x** **or** **y** vaut **False** seulement quand **x** et **y** valent **False**
  - **not** c'est la **négation** logique, **not** (**True**) = **False** et **not** (**False**) = **True**
- Les comparaisons produisent des booléens :
  - Le test d'égalité se fait avec **==**
  - Le test de différence se fait avec **!=**
  - On a aussi $<$ $<=$ $>$ $>=$ pour comparer selon l'ordre usuel (ordre du dictionnaire pour les chaînes)

Encore des exemples . . .

– Opérations sur les chaînes de caractères –

- Si on utilise **+** sur deux chaînes de caractères, on effectue la concaténation des deux chaînes :
  'IUT' + 'info' $\rightarrow$ **'IUTinfo'**
- Si on "multiplie" une chaîne par un entier **n**, on la répère **n** fois :
  'IUT' * 3 $\rightarrow$ **'IUTIUTIUT'**

– Autres opérations –

- Il existe beaucoup d'autres opérations sur les chaînes
- On a accès à plein d'opérations mathématiques (cosinus, . . .)
- On verra ça plus tard dans le semestre

– Nommage –

Dans le programme d'introduction, on a utilisé nos propres noms, en bleu :

```
def puissance(x,n):
    ...
y = puissance(12,6)
print(y)
```

Les **règles** de nommage pour ce cours sont les suivantes :

- le caractère "underscore" _ (le tiret bas de la touche 8) est considéré comme une lettre
- on n'utilise jamais d'accent, de cédille, ...
- Les noms commencent par une lettre majuscule ou minuscule, puis sont composés de lettres et de nombres :
  exemple _ex2 Ex2mpl1 ~~2013iut~~
- les mots réservés de Python sont interdits
- il y a aussi des **conventions**, *plus tard* ...

– Mots réservés –

Les mots suivants sont réservés pour le langage :

and	**as**	**assert**	**break**	**class**	**continue**
def	**del**	**elif**	**else**	**except**	**finally**
for	**from**	**global**	**if**	**import**	**in**
is	**lambda**	**nonlocal**	**not**	**or**	**pass**
raise	**return**	**try**	**while**	**with**	**yield**

- On n'utilisera pas non plus comme nom :**True** , **False** , **None**
- Pour voir la liste des mots réservés, dans un terminal Python taper :
  import keyword
  print(keyword.kwlist)

– Variables –

- une variable est un nom qui référence une valeur dans la mémoire
- on peut s'en servir dans les calculs
- elle a le même type que la valeur qu'elle référence

– Affectation –

- L'affectation d'une variable consiste à lier un nom à une valeur
- La syntaxe : **nom = expression**, où **expression** est une valeur ou un calcul qui produit une valeur :
  x = 3 **y = 'IUT'** **z = x + 2**
- On peut affecter à nouveau une même variable, on perd le lien avec l'ancienne valeur

Ce n'est pas du tout le = des mathématiques. Il faut le lire comme "prend la valeur" : **x = x + 1**

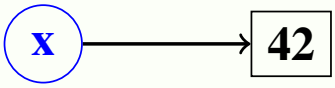
– Etapes de l'affectation –

x = 40 + 2

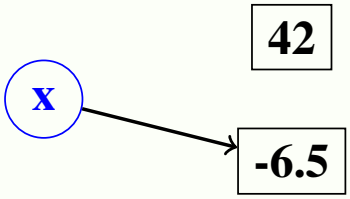
- On commence par calculer le membre droit, ici on trouve **42**

- Ensuite on crée le nom pour **x** (sauf s'il a déjà été créé)

- Enfin on relie la variable à sa valeur



- En cas de réaffectation, le lien d'avant est perdu : **x= -6.5**



Python pas à pas

Instructions et blocs

– Instructions et séquence d'instructions –

```
print('a x^2 + b x + c = 0')
a = float(input('a = '))
b = float(input('b = '))
c = float(input('c = '))
delta = b*b-4*a*c
if delta > 0:
    print('deux solutions')
elif delta == 0:
    print('une solution')
else:
    print('pas de solution')
```

- Comme on a vu dans l'introduction, les instructions sont effectuées dans l'ordre, de haut en bas
- En Python, il n'y a **qu'une instruction par ligne**
- Le flot d'instructions peut-être modifié / redirigé par des conditions (**if**), des boucles (**while**), ...

– Au passage ... input –

```
a = float(input('a = '))
```

- On a utilisé une nouvelle fonction, la fonction **input**(**str**)
- Cette fonction permet à l'utilisateur de saisir une valeur au clavier
- Quand on écrit **a= input('valeur = ')**, la chaîne **'valeur = '** est affichée à l'écran (comme avec **print**) et le programme attend que soit rentré une valeur, qu'il met dans la variable **x**, c'est une affectation normale
- La fonction **input** renvoie toujours une chaîne de caractères
- On a donc utilisé le transtypage avec la fonction **float**

Quelques exemples avec input

– Blocs d'instructions –

Certaines instructions sont regroupées en blocs de la façon suivante :

```
entête du bloc:
        instruction 1 du bloc
        instruction 2 du bloc
        instruction 3 du bloc
instruction hors bloc
```

- L'indentation (le décalage) se fait avec la tabulation (la touche au-dessus du capslock sur le clavier, cf TP)
- On peut insérer un bloc dans un bloc, un bloc dans un bloc dans un bloc, ...
- L'indentation fait partie du langage Python, changer l'indentation **change la signification** du programme

Python pas à pas

Instruction conditionnelle (if)

– La conditionnelle : le if –

```
if delta > 0:
    print('deux solutions')
elif delta == 0:
    print('une solution')
else:
    print('pas de solution')
```

- Sur l'exemple on commence par tester si **delta**$>$ **0**
  - Si c'est le cas, on effectue le bloc qui suit, et on affiche **deux solutions**
- Sinon, on teste si **delta**$==$ **0**
  - Si oui, on indique qu'il y a une seule solution
- Sinon on indique qu'il n'y a pas de solution

– La conditionnelle : le if –

La forme la plus simple est

```
if expression:
        instruction 1 du if
        instruction 2 du if
        ...
instruction après if
```

- **expression** est une expression qui retourne un booléen, qui est donc évaluée à **True** ou **False**
- les instructions du bloc du **if** sont effectuées **uniquement si** l'expression est évaluée à **True**
- dans tous les cas, le programme reprend à **l'instruction après if**

– La conditionnelle : le if avec else –

La forme avec **else** (= sinon) :

```
if  expression:
        instruction 1 du if
        ...
else:
        instruction 1 du else
        ...
instruction après if/else
```

- les instructions du bloc du **if** sont effectuées **uniquement si** l'expression est évaluée à **True**
- les instructions du bloc du **else** sont effectuées **uniquement si** l'expression est évaluée à **False**
- dans tous les cas, le programme continue à **l'instruction après if/else**

– La conditionnelle : le elif –

La forme avec **elif** (= contraction de else et if) :

if expression1:
    bloc du if
elif expression2:
    bloc du elif
else:
    bloc du else
instruction après if/elif/else

- les instructions du bloc du **if** sont effectuées **uniquement si** **expression1** vaut **True**
- les instructions du bloc du **elif** sont effectuées **uniquement si** **expression1** vaut **False** et **expression2** vaut **True**
- les instructions du bloc du **else** sont effectuées **uniquement si** **expression1** vaut **False** et **expression2** vaut **False**

– La conditionnelle : le elif –

La forme avec **elif** (= contraction de else et if) :

if expression1:
bloc du if
elif expression2:
bloc du elif
else:
bloc du else
instruction après if/elif/else

- les instructions du bloc du **if** sont effectuées **uniquement si expression1** vaut **True**
- les instructions du bloc du **elif** sont effectuées **uniquement si expression1** vaut **False** et **expression2** vaut **True**
- les instructions du bloc du **else** sont effectuées **uniquement si expression1** vaut **False** et **expression2** vaut **False**
- On peut mettre plusieurs **elif**, les conditions sont évaluées **dans l'ordre**, et seule **la première** qui vaut **True** est considérée

Attention Python 2 ≠ Python 3

Les deux versions **ne sont pas** compatibles !

Installez la version 3.1 ou 3.2

Introduction à l'algorithmique et à la programmation

IUT 1ère année 2013-2014

Cyril Nicaud

Cyril.Nicaud@univ-mlv.fr

– Cours 2 / 5 –

Graphiques

La librairie `iutk`

– Présentation générale –

- **iutk** est une librairie développée pour vous
- elle permet de faire des graphiques et des animations visuelles
- c'est juste une surcouche simplifiée d'une librairie Python nommée **tkinter**

- Que peut-on faire ?

– Présentation générale –

- **iutk** est une librairie développée pour vous
- elle permet de faire des graphiques et des animations visuelles
- c'est juste une surcouche simplifiée d'une librairie Python nommée **tkinter**

- Que peut-on faire ?
  - ouvrir une fenêtre
  - dessiner des cercle, des rectangles, du texte, ...
  - gérer des couleurs
  - gérer des événements de la souris ou du clavier
- Pourquoi ne pas utiliser **tkinter** directement ?

– Présentation générale –

- **iutk** est une librairie développée pour vous
- elle permet de faire des graphiques et des animations visuelles
- c'est juste une surcouche simplifiée d'une librairie Python nommée **tkinter**

- Que peut-on faire ?
  - ouvrir une fenêtre
  - dessiner des cercle, des rectangles, du texte, ...
  - gérer des couleurs
  - gérer des événements de la souris ou du clavier
- Pourquoi ne pas utiliser **tkinter** directement ?
  - **tkinter** utilise des notions de programmation objet qui ne sont pas au programme de ce cours
  - **iutk** sert justement à cacher les aspects "objet"
- Est-ce que **iutk** permet de faire la même chose que **tkinter** ?

– Présentation générale –

- **iutk** est une librairie développée pour vous
- elle permet de faire des graphiques et des animations visuelles
- c'est juste une surcouche simplifiée d'une librairie Python nommée **tkinter**

- Que peut-on faire ?
  - ouvrir une fenêtre
  - dessiner des cercle, des rectangles, du texte, ...
  - gérer des couleurs
  - gérer des événements de la souris ou du clavier
- Pourquoi ne pas utiliser **tkinter** directement ?
  - **tkinter** utilise des notions de programmation objet qui ne sont pas au programme de ce cours
  - **iutk** sert justement à cacher les aspects "objet"
- Est-ce que **iutk** permet de faire la même chose que **tkinter** ?
  - Non, il y a plus de possibilités dans **tkinter**, mais **iutk** nous suffira pour ce semestre

– Exemple 1 : créer une fenêtre –

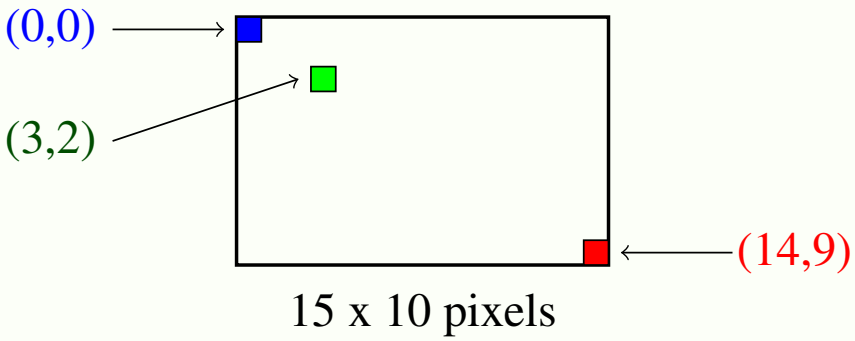
```
from iutk import *

creeFenetre(400,600)
attenteClic()
fermeFenetre()
```

- la première ligne sert à appeler la bibliotheque **iutk**
- on crée la fenêtre avec la fonction **creeFenetre(... , ...)**, en indiquant la hauteur et la largeur en nombre de **pixels**
- **attenteClic()** attend que l'utilisateur clique dans la fenêtre avant de continuer
- **fermeFenetre()** détruit la fenêtre à la fin du programme (ne pas oublier de le faire).

– Fenêtre et pixels –

- Un écran ou une fenêtre est un objet à deux dimensions
- Ce sont des “damiers” dont les cases sont des points (carrés) de couleur appelés des **pixels**
- 1600 x 1200 est un standard pour un écran, 640 x 960 pour un iphone 4s
- les pixels ont un système de **coordonnées cartésiennes**
- **Attention :** le coin en haut à gauche est de coordonnées $(0, 0)$
- les coordonnées augmentent vers le bas et vers la droite



– Lignes, rectangles et cercles –

On a des commandes pour dessiner dans la fenêtre :

- des segments avec **ligne** et **ligneCouleur**, en donant les coordonnées des deux extrémités et éventuellement la couleur
- des rectangles avec **rectangle**, **rectangleCouleur** et **rectanglePlein**, en donnant les coordonnées de deux coins opposés, et éventuellement la couleur
- des rectangles avec **cercle**, **cercleCouleur** et **cerclePlein**, en donnant les coordonnées de deux coins opposés, et éventuellement la couleur

– Lignes, rectangles et cercles –

On a des commandes pour dessiner dans la fenêtre :

- des segments avec **ligne** et **ligneCouleur**, en donant les coordonnées des deux extrémités et éventuellement la couleur
- des rectangles avec **rectangle**, **rectangleCouleur** et **rectanglePlein**, en donnant les coordonnées de deux coins opposés, et éventuellement la couleur
- des rectangles avec **cercle**, **cercleCouleur** et **cerclePlein**, en donnant les coordonnées de deux coins opposés, et éventuellement la couleur

– Affichages –

- **miseAJour()** à appeler après avoir dessiné, sans quoi le résultat est incertain
- **effaceTout()** enlève tous les dessins

– Lignes, rectangles et cercles –

On a des commandes pour dessiner dans la fenêtre :

- des segments avec **ligne** et **ligneCouleur**, en donant les coordonnées des deux extrémités et éventuellement la couleur
- des rectangles avec **rectangle**, **rectangleCouleur** et **rectanglePlein**, en donnant les coordonnées de deux coins opposés, et éventuellement la couleur
- des rectangles avec **cercle**, **cercleCouleur** et **cerclePlein**, en donnant les coordonnées de deux coins opposés, et éventuellement la couleur

– Affichages –

- **miseAJour()** à appeler après avoir dessiné, sans quoi le résultat est incertain
- **effaceTout()** enlève tous les dessins

– Autres –

Voir la documentation pour les autres fonctionalités, notamment la récupération d'événements clavier et souris.

– Un exemple –

```
from iutk import *

creeFenetre(400,200)

ligne(10,10,300,100)
cercleCouleur(50,50,10,'blue')
rectanglePlein(250,40,350,100,'red')

miseAJour()
attenteClic()
fermeFenetre()
```

A tester ...

Python pas à pas

Utilisation des modules

– Modules –

- un **module** est un ensemble de fonctions déjà programmées, prêtes à être utilisées
- il existe des **blue modules de base** en python (`math` par exemple), mais vous pouvez en récupérer sur internet ou en créer vous-même
- utilisé pour organiser les programmes (on en reparlera)

Il y a deux façons d'utiliser une fonction **toto()** d'un module **monModule**:

- **from monModule import toto** : on utilise alors normalement la fonction
- **import monModule** : la fonction s'appelle **monModule.toto()**

– Modules –

- un **module** est un ensemble de fonctions déjà programmées, prêtes à être utilisées
- il existe des **blue modules de base** en python (`math` par exemple), mais vous pouvez en récupérer sur internet ou en créer vous-même
- utilisé pour organiser les programmes (on en reparlera)

Il y a deux façons d'utiliser une fonction **toto()** d'un module **monModule**:

- **from monModule import toto** : on utilise alors normalement la fonction
- **import monModule** : la fonction s'appelle **monModule.toto()**
- on peut aussi utiliser **from monModule import *** pour charger toutes les fonctions du module **monModule**, sauf celles qui commencent par _

– Un exemple –

```
from math import *

print('pi = ',pi)
print(cos(pi/2))
```

– Quelques modules utiles –

- **random** : des fonctions pour faire des tirages au sort
- **math** : les fonctions et les constantes mathématiques usuelles comme exp, cos, π, ...
- **time** : pour mesurer le temps, connaître l'heure, ou attendre un certain temps
- **iutk !**

Python pas à pas

La boucle `while`

– Exemple –

```
res = 1
i = 1
while i < n:
    res = res * x
    i = i + 1
print(res)
```

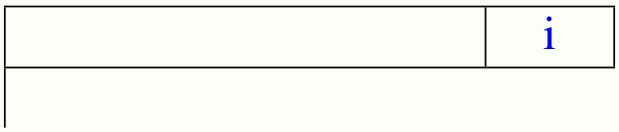
- on avait vu le **while** lors du cours 1
- c'est une **instruction de boucle** : son bloc associé est répété tant que la **condition est vraie**

while condition:
    instruction 1 du while
    instruction 2 du while
    ...
instruction après while

- chaque exécution de la séquence d'instructions du **while** est appelé une **itération**

– Utilisation : faire un nombre fixé de fois –

```
i = 1
while i <= n:
        instruction 1 du while
        instruction 2 du while
        ...
        i = i + 1
instruction après while
```



– Utilisation : faire un nombre fixé de fois –

```
i = 1
while i <= n:
        instruction 1 du while
        instruction 2 du while
        ...
        i = i + 1
instruction après while
```

	i
Avant de commencer	1

– Utilisation : faire un nombre fixé de fois –

```
i = 1
while i <= n:
        instruction 1 du while
        instruction 2 du while
        ...
        i = i + 1
instruction après while
```

	i
Avant de commencer	1
Fin itération 1	2

– Utilisation : faire un nombre fixé de fois –

```
i = 1
while i <= n:
        instruction 1 du while
        instruction 2 du while
        ...
        i = i + 1
instruction après while
```

	i
Avant de commencer	1
Fin itération 1	2
Fin itération 2	3

– Utilisation : faire un nombre fixé de fois –

```
i = 1
while i <= n:
        instruction 1 du while
        instruction 2 du while
        ...
        i = i + 1
instruction après while
```

	i
Avant de commencer	1
Fin itération 1	2
Fin itération 2	3
...	

– Utilisation : faire un nombre fixé de fois –

```
i = 1
while i <= n:
        instruction 1 du while
        instruction 2 du while
        ...
        i = i + 1
instruction après while
```

	i
Avant de commencer	1
Fin itération 1	2
Fin itération 2	3
...	
Fin itération n-1	n

– Utilisation : faire un nombre fixé de fois –

```
i = 1
while i <= n:
        instruction 1 du while
        instruction 2 du while
        ...
        i = i + 1
instruction après while
```

	i
Avant de commencer	1
Fin itération 1	2
Fin itération 2	3
...	
Fin itération n-1	n
Fin itération n	n+1

– Deux exemples –

```
nbr = int(input('Combien ? '))
i = 1
while i <= nbr:
    print('bonjour')
    i = i + 1
print('Fini !')
```

– Deux exemples –

```
nbr = int(input('Combien ? '))
i = 1
while i <= nbr:
    print('bonjour')
    i = i + 1
print('Fini !')
```

```
from time import sleep
nbr = int(input('Combien de secondes ? '))
i = nbr
while i > 0:
    print(str(i) + '...')
    sleep(1)
    i = i - 1
print('BOOOM !')
```

– Un exemple graphique –

```
from iutk import *
creeFenetre(400,400)
i = 0
while i < 10:
    if i % 2 == 1:
        couleur = 'blue'
    else:
        couleur = 'yellow'
    rectanglePlein(i*40,100,(i+1)*40,140,couleur)
    i = i + 1
miseAJour()
attenteClic()
fermeFenetre()
```

- On notera l'utilisation du reste de la division pour alterner les couleurs
- Les pavés font 40 pixels de côtés, on calcule les coordonnées de chaque carré en fonction de **i**

– Nombre d’itérations non fixé –

Redemander un nombre tant que nécessaire :

– Nombre d'itérations non fixé –

Redemander un nombre tant que nécessaire :

```
a = int(input('Nombre entre 1 et 10 : '))
while a < 1 or a > 10:
    a = int(input('Erreur, entre 1 et 10 :'))
print('Bravo, votre nombre est ' + str(a))
```

Combien de tirages de deux dés pour faire 12 :

– Nombre d'itérations non fixé –

Redemander un nombre tant que nécessaire :

```
a = int(input('Nombre entre 1 et 10 : '))
while a < 1 or a > 10:
    a = int(input('Erreur, entre 1 et 10 :'))
print('Bravo, votre nombre est ' + str(a))
```

Combien de tirages de deux dés pour faire 12 :

```
from random import randint
des = 0
nbr = 0
while des != 12:
    des = randint(1,6) + randint(1,6)
    nbr = nbr + 1
print(str(nbr)+' tirages pour faire 12')
```

– Un peu de physique –

- On veut simuler la chute d'une goutte d'eau
- La goutte commence à vitesse nulle et tombe sous l'effet de la gravité
- Elle a une accélération constante $\vec{g}$, qui est un vecteur dirigé vers le bas

– Un peu de physique –

- On veut simuler la chute d'une goutte d'eau
- La goutte commence à vitesse nulle et tombe sous l'effet de la gravité
- Elle a une accélération constante $\vec{g}$, qui est un vecteur dirigé vers le bas
- L'accélération est la dérivée de la vitesse : à "chaque instant de temps", la vitesse verticale diminue de g
- A "chaque instant de temps", la position verticale de la goutte diminue de sa vitesse

– Un peu de physique –

- On veut simuler la chute d'une goutte d'eau
- La goutte commence à vitesse nulle et tombe sous l'effet de la gravité
- Elle a une accélération constante $\vec{g}$, qui est un vecteur dirigé vers le bas
- L'accélération est la dérivée de la vitesse : à "chaque instant de temps", la vitesse verticale diminue de g
- A "chaque instant de temps", la position verticale de la goutte diminue de sa vitesse
- On va **identifier** "itération de boucle" et "instant de temps"
- Il faut garder la vitesse verticale et la position verticale en mémoire (la position horizontale ne change pas)
- A chaque itération on met à jour les deux variables
- **Attention :** dans la fenêtre les ordonnées augmentent vers le bas !
- On recalcule le graphique en fonction des nouvelles valeurs
- On s'arrête quand on touche le sol

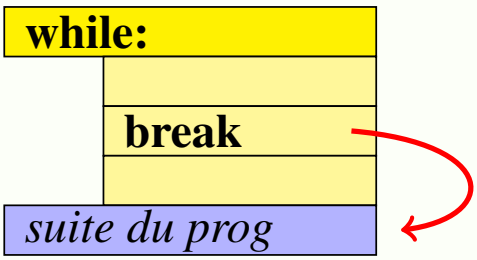
– La goutte d'eau –

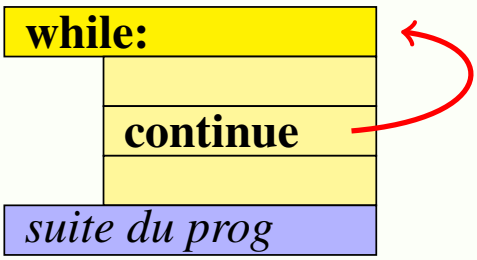
```
from iutk import *
creeFenetre(400,400)
attenteClic()
y = 0
vy = 0
while y < 400: # on n'a pas encore touche le sol
    effaceTout()
    cerclePlein(200,y,10,'blue')
    miseAJour()    # ne pas oublier !
    vy = vy + 0.1  # ici g = 0.1
    y = y + vy     # mise a jour de la position
attenteClic()
fermeFenetre()
```

- Il faut ajuster la valeur de g pour avoir un bon rendu

– break et continue –

- Il existe deux moyens de modifier le flot normal des itérations d'une boucle
- Ces moyens sont à éviter, sauf quand ils rendent le programme plus clair (ce qui arrive)
- l'instruction **break** arrête la boucle, et continue donc l'exécution du programme après la boucle
- l'instruction **continue** arrête l'itération en cours, et reprend à l'itération suivante de la boucle





– Un exemple avec break –

```
from random import randint
de = randint(1,100)
while True:
    a = int(input('Devinez un nombre (1-100) : '))
    if a == de:
        break
    if a < de:
        print('trop petit')
    if a > de:
        print('trop grand')
print('Bravo !')
```

- **while True** boucle indéfiniment, la condition est toujours vérifiée
- On utilise **break** pour arrêter les itérations quand le nombre est deviné

– Boucles infinies –

Si la condition de boucle est toujours vraie, et qu'il n'y a pas de **break** pour en sortir, le programme **reste bloqué en tournant indéfiniment dans la boucle !** C'est une erreur classique.

- Pour forcer l'arrêt d'un programme, dans `idle` ou un terminal python faire control+C
- Pour forcer la fermeture faire control+D

– Les boucles imbriquées –

- On peut utiliser des boucles dans des boucles, des boucles dans des boucles ...
- Cela arrive fréquemment quand on veut gérer des objets à deux dimensions, mais aussi dans d'autres situations (voir les algorithmes de tri dans l'autre cours)

```
i = 1
while i < 5:
    j = 1
    while j < 4:
        print('('+str(i)+','+str(j)+')')
        j = j + 1
    i = i + 1
```

– Exemple du damier –

On souhaite paver la fenêtre de carrés colorés :

– Exemple du damier –

On souhaite paver la fenêtre de carrés colorés :

```
from iutk import *
creeFenetre(400,400)
i = 0
while i < 10:
    j = 0
    while j < 10:
        if (i+j) % 2 == 1:
            couleur = 'blue'
        else:
            couleur = 'yellow'
        rectanglePlein(i*40,j*40,(i+1)*40, ...
        j = j + 1
    i = i + 1
miseAJour()
attenteClic()
fermeFenetre()
```

Python pas à pas

Les listes

– Présentation –

- La **liste** est un **type avancé de données**, elle sert à stocker une séquence de valeurs
- On peut créer une liste par une affectation normale, où on met entre crochets et séparés par des virgules les différentes valeurs de la liste

lst = [3, 'toto', 4.5, False]

- Il y a une liste particulière, la **liste vide** **[]** qui ne contient aucun élément
- On peut accéder au *i*-ème élément d'une liste en utilisant les crochets, le *i*-ème élément de **lst** est **lst[i]**
- **Attention :** les indices commencent à 0 et non à 1 !

```
lst = [ 3, 'toto', 4.5, False ]
print( lst[1] )
>>> 'toto'
```

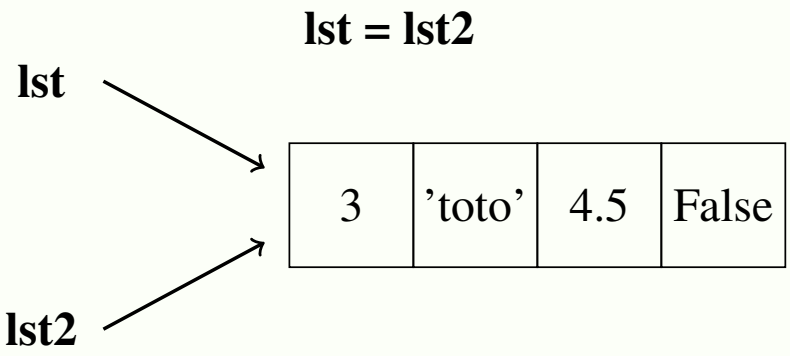
– Affectations et listes –

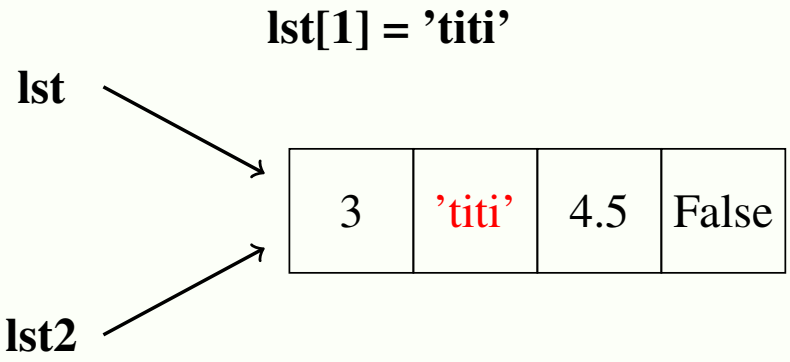
Il y a des subtilités ici, à bien travailler.

- Une liste est un objet **modifiable**, on peut modifier ses valeurs, ses éléments, *etc* sans créer de nouvelle liste
- Ce n'est pas vrai pour les autres types qu'on a vu, notamment les chaînes de caractères
- On peut changer la *i*-ème valeur de **lst** avec une affectation classique : **lst[2] = 'titi'**
- Si **lst** est une liste, **lst2 = lst** associe au nom **lst2** la même liste (qui est modifiable).
- En conséquence, si on modifie ensuite **lst**, **on modifie aussi lst2**!

```
lst = [ 3, 'toto', 4.5, False ]
lst2[1] = 'titi'
print(lst)
>>> [ 3, 'titi', 4.5, False ]
```

– Affectations et listes –





– Quelques opérations –

- **len(lst)** retourne la longueur de **lst**
- **x in lst** renvoie un booléen qui est **True** quand **x** est dans **lst**
- Le **+** crée une **nouvelle liste** qui est la concaténation de deux listes

```
[ 3, 4 , 7] + ['toto', 5]
>>> [ 3, 4, 7, 'toto', 5]
```

- Si on multiplie une liste par un entier *n*, cela crée une **nouvelle liste** où l'ancienne est répétée *n* fois :

```
lst = [ 3, 4 , 7]
lst2 = lst * 3
print(lst2)
>>> [ 3, 4, 7, 3, 4, 7, 3, 4, 7]
```

– Autres opérations –

- Les liste sont des **objets**, une notion hors-programme. On peut néanmoins utiliser certaines fonctions rattachées aux listes (on appelle cela des méthodes)
- Pour utiliser une telle fonction sur une liste **lst**, on utilise **lst.nom()**
- **Attention :** en générale cela **modifie** la liste

Quelques exemples :

- **lst.append(x)** ajoute la valeur de **x** à la fin de **lst**
- **lst.extend(lst2)** ajoute tous les éléments de **lst2** à la fin de **lst**
- **lst.pop()** supprime le dernier élément de **lst** et retourne sa valeur
- **lst.pop(i)** supprime le i-ème élément de **lst** et retourne sa valeur
- ...

– Parcourir une liste –

- Une façon naturelle de **parcourir** une liste est d'utiliser une boucle **while**, en faisant varier l'indice dans la liste

```
lst = [1, 'toto', 4.5, False]
i = 0
while i < len(lst):
    print(lst[i])
    i = i + 1
```

- Il existe d'autre moyens de parcourir une liste, que l'on verra au prochain cours

– Exemple : statistiques sur deux dés –

- On jette deux dés et on fait la somme. Comment se répartissent les différents tirages ?

– Exemple : statistiques sur deux dés –

- On jette deux dés et on fait la somme. Comment se répartissent les différents tirages ?

```
lst = [1, 'toto', 4.5, False]
i = 0
while i < len(lst):
    print(lst[i])
    i = i + 1
```

– Exemple : paradoxe des anniversaires –

- A partir de combien de personnes dans une pièce des chances importantes qu'au moins deux aient la même date d'anniversaire ?
- On va simuler, en regardant plusieurs fois au bout de combien de personnes on a un doublon

– Exemple : paradoxe des anniversaires –

- A partir de combien de personnes dans une pièce des chances importantes qu'au moins deux aient la même date d'anniversaire ?
- On va simuler, en regardant plusieurs fois au bout de combien de personnes on a un doublon

```
from random import randint
lst = [] # pas de dates au debut
compteur = 0
while True:
    compteur = compteur + 1
    annee = randint(1,365)
    if annee in lst: #doublon
        break
    lst.append(annee)
print(compteur,'personnes')
```

Introduction à l'algorithmique et à la programmation

IUT 1ère année 2013-2014

Cyril Nicaud

Cyril.Nicaud@univ-mlv.fr

– Cours 3 / 5 –

– Ecriture condensée –

- On a souvent besoin d'ajouter une valeur dans une variable, ce que l'on a fait avec **x = x + y**
- Il existe en Python (et dans beaucoup d'autre langages) une écriture plus compacte pour faire la même chose : **x += y**
- On peut l'utiliser avec d'autres opérations, et sur différents type. Pour **x** entier et **s** chaîne de caractères, on a :

x += 3	$\longrightarrow$	ajoute 3 à **x**
x *= 2	$\longrightarrow$	multiplie **x** par 2
x //= 4	$\longrightarrow$	**x** est changé en son quotient par 4
s += 'toto'	$\longrightarrow$	concatène **'toto'** à la fin de `s`
s *= 3	$\longrightarrow$	remplace **s** par 3 copies de `s`
s //= 4	$\longrightarrow$	erreur

- *il n'y a pas de notation* `i++` *en Python*

Python pas à pas

Les chaînes de caractères

– Déclaration de chaînes de caractères –

- On peut déclarer une chaîne entre apostrophes comme on a fait jusqu'ici : `x = 'toto'` ...
- ou entre guillemets : `x = "toto"`
- les deux sont valides, on peut par exemple utiliser la première quand il y a des guillemets dans la chaîne et la seconde quand il y a des apostrophes.

- Comment faire s'il y a à la fois des ' et des " ? on utilise les caractères spéciaux \' et \" :

```
s = 'il a dit :  "à l\'abordage !"'
```

- **Attention :** \' est un seul caractère, de même pour \" (ce sont des caractères spéciaux) :

```
len('d\'abord')    ⟶    7
```

– Déclaration sur plusieurs lignes –

- On peut déclarer une chaîne sur plusieurs lignes en utilisant des triples apostrophes ou triples guillemets comme délimiteurs :
  s = ”’Ceci est une
  chaine sur
  plusieurs lignes.”’
- Les saut de lignes seront encodés par le caractère \n
- On peut également utiliser juste un backslash \avant la fin de ligne et continuer sur la ligne suivante **s = ’Ceci est une**\
  chaine sur\
  plusieurs lignes.’

– Caractères spéciaux –

- Voilà quelques caractères spéciaux utiles :

\’	apostrophe	\”	guillemet
\n	saut de ligne	\t	tabulation
\\	antislash	\a	reculer d’un

- Par exemple la chaîne `x = 'toto\ba'` est une chaîne de longueur 6, si on fait `print(x)` il s’affiche ...

```
>>> print(x)
tota
```

– Les chaînes sont non-modifiables –

- **Important :** une chaîne n'est pas modifiable.
- Si **x** contient une valeur de type **str** et que vous voulez la changer, il faut faire une nouvelle affectation de **x** :

  x = 'toto'
  x[0] = 'p' $\longrightarrow$ **erreur** on ne peut pas modifier une chaîne
  x = 'poto' $\longrightarrow$ on crée une nouvelle chaîne 'poto'

- **Rappel :** c'est le contraire avec les listes :

  lst = [1,4,6,7]
  lst[0] = 3 $\longrightarrow$ **lst** vaut **[3,4,6,7]**

– Opérations sur les chaînes –

- On a déjà vu la concaténation **+** de deux chaînes et la "multiplication" par un entier
- On a accès au *i*-ème caractère de la chaîne **s** avec **s[i]** (les indices commencent à 0)
- **len(s)** retourne la longueur de **s**

Il y a beaucoup d'autres opérations sur les chaines, avec la notation **s.fonction()** notamment :

- **s.lower()** renvoie une nouvelle chaîne où les majuscules ont été changées en minuscules
- **s.upper()** renvoie une nouvelle chaîne où les minuscules ont été changées en majuscules
- **s.split(t)**, où **t** est une chaîne, renvoie un tableau de chaînes obtenues en coupant **s** aux occurences de **t**
- ...

Python pas à pas

Structures itérables et boucles for

– Structure itérable –

- Une **structure itérable** est une structure qui contient plusieurs valeurs avec
  - une valeur initiale
  - une notion de valeur suivante

- On connaît déjà deux exemples de structures itérables : les chaînes de caractères et les listes :
  s = ”abcdef”
  lst = [1, 4, 56, 2]

- On peut changer un itérable en la liste, dans l’ordre, de ses élément avec l’instruction **list( )**

– les range –

- Une autre structure itérable très utilisée est retournée par la fonction **range( )**
- **range(a,b)**, où **a** et **b** sont des entiers, est un itérable qui commence à **a** et qui s'arrête à **b-1** :
  list(range(1,5)) $\longrightarrow$ **[1,2,3,4]**
- **range(b)** est une version condensée de **range(0,b)**
- **range(a,b,c)** est l'itérable qui commence à **a** et avance de **c** en **c** jusqu'à arriver en **b** (exclu)
  list(range(1,7,2)) $\longrightarrow$ **[1,3,5]**

- **attention** on s'arrête avant **b** dans tous les cas

– les boucles for –

- Comme **while**, l'instruction **for** est une instruction de boucle
- Elle permet de parcourir un itérable, dans l'ordre, en commençant au premier élément et en allant de suivant en suivant
- La syntaxe est la suivante

for x in *iterable*:
    instruction 1 du for
    instruction 2 du for
    ...
    instruction n du for
suite du programme

– Faire une action *n* fois –

- l'association de **for** et de **range** rend très facile de faire une opération *n* fois :

```
n = int(input('rentrez un nombre : '))
for i in range(n):
    print('bonjour')
```

- **i** prend les valeurs du **range**, à savoir **0**,**1**,. . . **n-1**
- Autre exemple : les statistiques sur la somme de deux dés

```
from random import randint
stats = [0] * 13
for i in range(1000):
    stats[randint(1,6)+randint(1,6)] += 1
print(stats)
```

– Utiliser la suite des valeurs d'un range –

- Afficher les nombres de 1 à n :

```
n = int(input('rentrez un nombre : '))
for i in range(n):
    print(i)
```

- Compte à rebours :

```
from time import sleep
for i in range(5,0,-1):
    print(str(i)+'...')
    sleep(1)
print('BOOOM')
```

– Itérer sur une liste –

- Afficher un à un les éléments d'une liste :

```
lst = [3,5,6,14,-6,121]
for x in lst:
    print(x)
```

- Changement de couleur :

```
lst = ['red','blue','green','gray','black']
for couleur in lst:
    effaceTout()
    cerclePlein(200,200,100,couleur)
    cercle(200,200,100)
    miseAJour()
    sleep(1)
attenteClic()
```

– Itérer sur les indices d'une liste –

- Si on a besoin des indices lors du parcours d'un itérable **iterable**, on peut utiliser **range(len(iterable))**, vu que les indices vont de **0** à **len(iterable)-1**

```
lst = ['bon','jour','bonjour']
for i in range(len(lst)):
    print(i,lst[i])
```

– Itérer sur une chaîne –

- Compter le nombre de voyelles :

```
s = input('texte : ')
nbrVoyelles = 0
for a in s.lower():
    if a in ['a','e','i','o','u','y']:
        nbrVoyelles += 1
print('il y a',nbrVoyelles,'voyelles')
```

- Jeu du pendu (extrait) :

```
motPendu = ''
for a in mot:
    if a in proposes: # c'est une lettre proposee?
        motPendu += a
    else:
        motPendu += '-'
```

– continue et break –

- On peut utiliser les instructions **continue** et **break** avec les boucles **for** :
  - **continue** reprend au **for** en passant à l'élément suivant de l'itérable
  - **break** interrompt la boucle

```
n = int(input('nombre : '))
for i in range(2,n):
    if n % i == 0:
        print(n,'n\'est pas premier')
        print('il est divisible par',i)
        break
```

– Conclusion sur la boucle for –

- On peut toujours faire une boucle **while** à la place ... c'est ce qu'on a fait jusqu'ici
- L'instruction **for** est pluscompacte, plus lisible, et donc souvent meilleure quand elle est utilisable
- Elle n'est typiquement pas adaptée quand on ne sait pas au début de la boucle combien de fois on va l'effectuer (ex: deviner un nombre)

Python pas à pas

Les fonctions

– Présentation générale –

```
def estPremier(n):
    if n < 2:
        return False
    for i in range(2,n):
        if n % i == 0:
            return False
    return True
```

- Une **fonction** est un bloc d'instruction réutilisable
- Cela permet d'écrire le code une seule fois pour réaliser une même tâche répétée :
  - Une fois bien testée, on s'en ressert autant qu'on veut
  - Maintenance à effectuer à un seul endroit
  - On peut mettre les fonctions dans un module pour les réutiliser
- Idée fondamentale en programmation : découper un programme en sous-tâches pour gagner en lisibilité et en robustesse.

– Définir une fonction –

```
def nomFonction():
    instruction 1 de la fonction
    ...
    fin du bloc de la fonction
```

- **Important :** lors de la définition d'une fonction, le code **n'est pas** exécuté

– Appeler une fonction –

- A tout moment dans le **programme** ou dans une **fonction** on peut appeler la fonction avec la commande **nomFonction()**

```
for i in range(2,100):
    if estPremier(i):
        print(i)
```

– Premier exemple –

```
def appel():
    print('-'*5,'appel',-'*5)

print('bonjour')
appel()
n = int(input('nombre = '))
for i in range(n):
    appel()
```

- A chaque fois qu'on utilise l'instruction **appel()** le programme interrompt le flot normal d'instructions pour aller effectuer les instructions d'**appel()**
- Une fois les instructions d'**appel()** effectuées, le programme reprend là où il en était

– Fonction avec paramètre –

```
def affiche(s):
    print('*' * (len(s)+4))
    print('* '+s+' *')
    print('*' * (len(s)+4))

affiche('bonjour')
texte = input()
affiche(texte)
```

- Une fonction peut avoir un ou plusieurs paramètres
- Ils sont nommés entre parenthèses dans la définition de la fonction
- Lorsque l'on appelle la fonction, il faut passer les paramètres (le bon nombre) entre parenthèses

– L'instruction return –

- l'instruction **return** **x** interrompt l'exécution de la fonction et retourne la valeur **x**
- **x** peut être de n'importe quel type
- On récupère la valeur retournée normalement, par exemple par une affectation :
  y = maFonction(x)
- On peut aussi l'utiliser dans une expression où elle est évaluée :
  y = maFonction(x) + 3
  print(maFonction(x))
- Par défault, s'il n'y a pas de **return** ou si on met **return** simplement sans argument après, la fonction retourne **None**

– Exemple de return –

```
def minimum(lst):
    if len(lst) == 0:
        return
    mini = lst[0] #on initialise a lst[0]
    for x in lst:
        if x < mini:
            mini = x
    return mini
```

- Le premier **return** n'a pas d'argument, il retourne **None** et arrête la fonction. Le programme reprend là où il en était.
- Le second **return** renvoie le résultat (flottant) du calcul

– Portée des variables –

Attention il y a des subtilités ici, à bien travailler.

```
def f(n):
    n = n + 1
x = 3
f(x)
print('x vaut',x)
```

- Le résultat est **x vaut 3**
- Ce qui se passe :
  - à l'appel de la fonction, la valeur du paramètre de **f** est affecté au **n** de la définition de **f**
  - donc **n** vaut **3**
  - dans la fonction, **n** est augmenté de 1
  - **x** n'a pas changé
  - ... d'ailleurs **n** n'existe pas dans le corps du programme

– Portée des variables –

Attention il y a des subtilités ici, à bien travailler.

```
def f(n):
    n = n + 1
n = 3
f(n)
print('n vaut',n)
```

- Le résultat est encore **n vaut 3 !**
- Ce qui se passe :
  - à l'appel de la fonction, la valeur du paramètre de **f** est affecté au **n** de la définition de **f**
  - Ce **n'est pas** le même **n**
  - Il y a le **n** principal, et le **n** de **f** qu'on va noter $\mathbf{n}_f$
  - $\mathbf{n}_f$ prend la valeur de **n** à l'appel de **f** et est incrémenté de 1 dans la fonction. **n** ne change pas.

– Portée des variables –

Attention il y a des subtilités ici, à bien travailler.

```
def f(x):
    n = 1
n = 3
f(n)
print('n vaut',n)
```

- Le résultat est toujours **n vaut 3 !**
- Ce qui se passe :
  - à l'appel de la fonction, la valeur du paramètre de **f** est affecté au **x** de la définition de **f**
  - L'affectation dans la fonction crée une variable **locale** à **f**, notons-là $\mathbf{n}_f$
  - $\mathbf{n}_f$ prend la valeur de 1 et le **n** principal ne change pas.

– Porté des variables –

- les paramètres de la définition de la fonction sont des variables locales, propres à la fonction
- les variable affectées dans la fonction sont des variables locales, propres à la fonction
- ces variables locales existent pendant l'exécution de la fonction et n'existent plus après

- les variables affectées dans le corps du programme (hors fonctions) sont des variables globales
- les variables globales sont lisibles dans tout le programme
- les variables globales ne sont pas modifiables dans une fonction
- (si on affecte une variable globale dans une fonction, on crée une variable locale avec le même nom)
- pour modifier une variable globale **x** dans une fonction, il faut la déclarer avec le mot clé **global**

– Exemple de portée –

```
def f(n):
    global k
    i = n
    k = 0
    print(i,j,k)
i = 2
j = 4
k = 6
f(44)
print(i,j,k)
```

- Dans le corps de la fonction **n** et **i** sont des variables locales
- **k** est une variable globale modifiable
- **j** est visible en tant que variable globale

– Exemple : balles rebondissantes –

- Une balle est donnée par 4 valeurs **[x,y,vx,vy]**, ses coordonnées et son vecteur vitesse.
- On va faire une fonction qui crée une nouvelle balle avec des stats aléatoires
- Une fonction pour dessiner une balle
- Une fonction pour déplacer une balle
- Dans le programme on crée la fenêtre, initialise une liste de balles, puis on répète déplacements et mises à jour

– Exemple : Poker fermé –

- On joue avec un jeu de 32 cartes
- On veut des fonctions pour créer un jeu, le mélanger, piocher une carte, piocher 5 cartes
- On veut tester s'il y a quelquechose de valeur dans le jeu (carré, full, couleur, ...)

– Cartes du poker sur un jeu de 32 –

0 = 7 pique	8 = 7 coeur	16 = 7 carreau	24 = 7 trefle
1 = 8 pique	9 = 8 coeur	17 = 8 carreau	25 = 8 trefle
2 = 9 pique	10 = 9 coeur	18 = 9 carreau	26 = 9 trefle
3 = 10 pique	11 = 10 coeur	19 = 10 carreau	27 = 10 trefle
4 = V pique	12 = V coeur	20 = V carreau	28 = V trefle
5 = D pique	13 = D coeur	21 = D carreau	29 = D trefle
6 = R pique	14 = R coeur	22 = R carreau	30 = R trefle
7 = As pique	15 = As coeur	23 = As carreau	31 = As trefle

Introduction à l'algorithmique et à la programmation

IUT 1ère année 2013-2014

Cyril Nicaud

Cyril.Nicaud@univ-mlv.fr

– Cours 4 / 5 –

Python pas à pas

Structures de données avancées

– tuple –

- Les **tuple** sont l'équivalent de la notion mathématique de n-uplets
- Déclaration **x = (4,3,1)** crée un tuple avec 3 entiers
- On peut aussi directement écrire **x = 4,3,1**
- Pour faire un **tuple** avec **un seul élément**, il faut utiliser une virgule : **x = (4,)**, sinon **x** est un **int** qui vaut 4
- On accède au i-ème élément d'un **tuple** comme pour les listes où les chaînes : **x[i]**, où les indices commencent à 0
- La longueur d'un tuple est retournée par la fonction **len()**
- On peut changer un **itérable** en tuple à l'aide de la fonction **tuple()** : **tuple(range(5))** $\Rightarrow$ **(0,1,2,3,4)**
- On peut concaténer deux **tuple** avec +

– tuple vs list –

Même s'ils se ressemblent, **tuple** et **list** sont des structures complètement différentes

- La principale différence c'est que :
  - Une **list** est **modifiable**
  - Un **tuple** **n'est pas modifiable**
- Si **t** est un tuple, **t[0] = 3** produit une erreur car **t** n'est pas modifiable
- Il n'y a pas de **append()** pour les **tuple**
- ...
- En fait, un **tuple** ressemble plsu à une chaîne qu'à une liste

– Set –

- **Set** en anglais signifie **ensemble**
- La structure **set** permet de gérer efficacement un ensemble de donnée
- Comme c'est un ensemble, chaque élément ne peut y être **violet qu'une seule fois**
- Comme c'est un ensemble, **l'ordre ne compte pas**
- Le mécanisme utilisé pour que cette structure soit efficace fait que les éléments d'un **set** doivent être **non modifiables**
- Un **set** est un objet modifiable

On ne peut mettre dans un **set** que des objets **non modifiables**, donc pas de **liste**, pas de **set** et pas de **dictionnaire**

– Opération sur les Set –

Dans le tableau, **s** et **t** sont des **set** et **l** est un **itérable** :

set()	crée un set vide	**len(s)**	longueur du set **s**
x in s	teste si $x \in s$	**x not in s**	teste si $x \notin s$
s <= t	teste si $s \subset t$	**s | t**	retourne $s \cup t$
s & t	retourne $s \cap t$	**s ^ t**	retourne $s \Delta t$
s.add(x)	ajoute x dans s	**s.remove(x)**	retire x de s
s.pop()	retourne et enlève un élément de s	**s.discard(x)**	comme **remove**, mais pas d'erreur si $x \notin s$
set(l)	crée un set avec **l**	**s==t**	teste l'égalité

- les **opérations en bleu** dans la table **modifient** le set **s**

– Test d'efficacité –

- On crée une **list** avec 10 000 entiers
- On teste si les 10 000 entiers sont dedans

- On fait la même chose avec un **set**
- On compare le temps d'exécution avec le module **time**

– Dictionnaires –

- Les **dictionnaires** permettent d'implanter de façon très efficace des fonctions (partielles)
- Dans certains langages, ils sont appelés des **tableaux associatifs**
- Cela permet d'associer à une **clé** une **valeur**
- Exemple d'utilisation :

```
D = {} # dictionnaire vide
D['toto'] = 4 # associe 4 a la cle 'toto'
D['titi'] = 6
print(D['toto']) # affiche 4
D['toto'] = 'bonjour' # remplace 4
print(D['toto']) # affiche 'bonjour'
print(D)
```

La clé doit être un élément **non modifiable**

– Opération sur les dictionnaires –

Dans le tableau suivant **D** est un dictionnaire, **x** est une clé et **y** est une valeur :

{}	dictionnaire vide	**len(D)**	nombre de clés
D[x] = y	D[x] vaut y	**D[x]**	retourne la valeur de x
del D[x]	x n'a plus de valeur	**x in D**	teste si x est une clé
D.keys()	la liste de clés	**D.values()**	la liste des valeurs
D.items()	la liste des couples (clé,valeur)		

– Modifiables et non-modifiables –

On a vu des structures **non-modifiables** :

- booléens, entiers, caractères, flottants
- chaînes de caractère
- tuple

Et des structures **modifiables** :

- listes
- ensembles
- dictionnaires

Il peut être utile de passer d'un type à l'autre. On peut par exemple utiliser la fonction **tuple()** pour transformer une liste (**modifiable**) en un tuple (**non-modifiable**)

– Affectation multiple –

- On peut affecter **simultanément** plusieurs variables avec la syntaxe **x,y,z = iterable**
- Cela ne fonctionne que s'il y a le **même nombre** d'éléments à gauche que dans l'itérable **x,y,z = [5,6,8]**
- Comme on peut **omettre les parenthèses** lors de l'écriture d'un tuple, on peut utiliser **x,y,z = 3,6,8**
- On peut même écrire **x,y = y,x** ce qui **échange les deux variables!**
- Il est possible d'utiliser _ pour signifer des positions qui ne nous intéressent pas

  x, _ , _ ,y = range(4)
- On peut s'en servir dans **toutes les situations**, par exemple

```
for key,value in D.items():
    print('cle=',key,'valeur=',value)
```

Python pas à pas

Compléments sur les fonctions

– Commentaire de fonction –

- Après l'entête de la fonction, on peut mettre un **descriptif de la fonction** directement dans une chaîne de caractères

```
def pgcd(a,b):
    'calcule le pgcd de a et de b'
    while b != 0:
        a, b = b, a % b
    return a
```

- On accède à la description avec la fonction **help** dans un terminal Python
- Certains éditeurs Python comme **IDLE3** font apparaître la description des fonctions

- Il faut prendre l'habitude de **mettre une description pour toute les fonctions importantes.**

– Paramètres par défaut –

- On peut **spécifier des valeurs par défauts** dans une fonctions

```
def f(x, y=4, z=5):
    return x + y + z
```

- Si les champs considérés ne sont pas donnés lors de l'appel à la fonction, ils prennent la valeur par défaut :
  f(2,2,2) $\rightarrow$ **6**
  f(2,3) $\rightarrow$ **f(2,3,5)** $\rightarrow$ **10**
  f(2) $\rightarrow$ **f(2,4,5)** $\rightarrow$ **11**
  f( ) $\rightarrow$ **erreur**

Il **ne faut pas** mettre des valeurs modifiables comme **valeurs par défaut**, mais vous pouvez mettre des **tuple, string, ...**

– Exemple –

```
def creeCercle(x=None,y=None,couleur=None):
    'par defaut le cercle est place au hasard'
    if x == None:
        x = randint(1,LARGEUR)
    if y == None:
        y = randint(1,HAUTEUR)
    if couleur == None:
        couleur = randomCouleur()
    cerclePlein(x,y,10,couleur)
```

– Paramètres modifiables –

Rappel :

```
def f(x):
    x = x + 1
n = 3
f(n)
print(n)
```

- le “x” de la fonction **f** est une variable locale de **f**, le “n” global n’est donc pas changé lors de l’appel à la fonction : cela affiche 3

Avec une liste :

```
def ajoute(L,x):
    L.append(x)
lst = [4,5]
ajoute(lst,7)
print(lst)
```

– Paramètres modifiables (suite) –

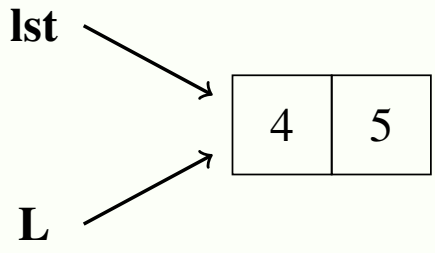
```
def f(x):
    x = x + 1 # x est incremente
n = 3
f(n)    # x de f prend la valeur de n
print(n)
```

```
def ajoute(L,x):
    L.append(x) # on modifie L en ajoutant x
lst = [4,5]
ajoute(lst,7) # L prend la valeur lst
print(lst) # affiche [4,5,7]
```

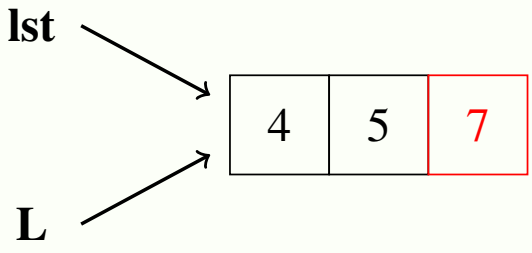
– Paramètres modifiables (suite) –

```
def ajoute(L,x):
    L.append(x) # on modifie L en ajoutant x
lst = [4,5]
ajoute(lst,7) # L prend la valeur lst
print(lst) # affiche [4,5,7]
```

ajoute(lst,7) → L,x = lst,7

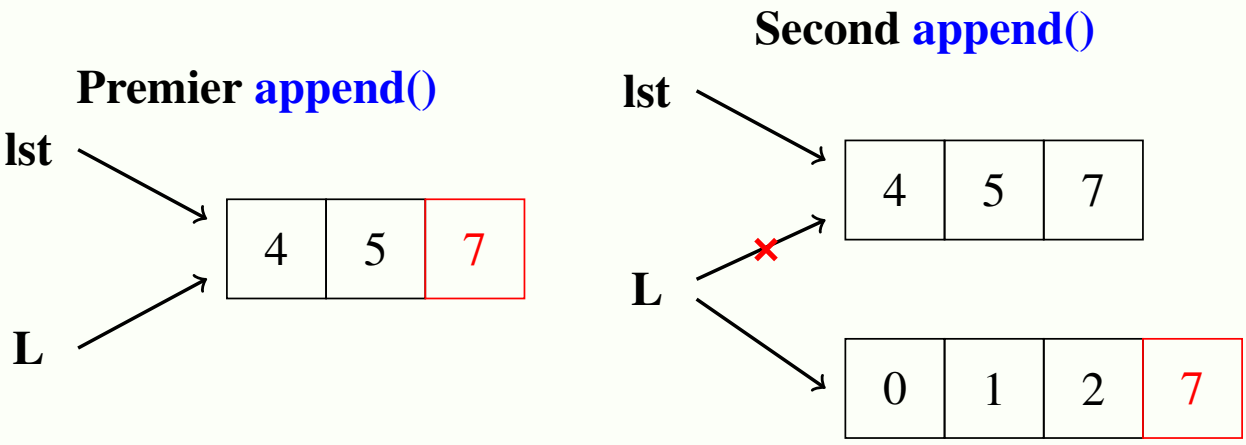


L.append(x)



– Paramètres modifiables (fin) –

```
def f(L,x):
    L.append(x)
    L = list(range(3))
    L.append(x)
lst = [4,5]
f(lst,7)
```



– Autre exemple : l'alphabet –

```
def ajouteLettres(u,D):
    for x in u:
        D.add(x)

A = set()
while True:
    s = input('mot = ')
    if s == 'stop':
        break
    ajouteLettres(s,A)
print('alphabet=',A)
```

– Fonctions en argument –

- On peut passer une **fonction en argument** d'une autre fonction

```
def filtre(L,f):
    R = []
    for x in L:
        if f(x):
            R.append(x)
    return R

def estPair(n):
    return n % 2 == 0

def toto(n):
    return n % 3 == 0

lst = filtre(range(10),estPair)
print(lst)
print(filtre(range(20),toto))
```

– Tri avec plusieurs fonctions de comparaison –

```
def triBulle(T,plusGrand):
    for i in range(len(T)-1,0,-1):
        for j in range(i):
            if plusGrand(T[j],T[j+1]):
                T[j], T[j+1] = T[j+1], T[j]

def superieurDebut(u,v):
    return u[0] > v[0]

def superieurFin(u,v):
    return u[len(u)-1] > v[len(v)-1]

def usuel(x,y):
    return x > y
```

– Exemple de A à Z: dessin d'une fonction –

- On va réaliser une fonction **trace(f,couleur,xmax,ymax)** pour afficher une fonction
- **f** est le nom de la fonction à tracer
- **couleur** est la couleur utilisée pour la tracer
- **xmax** et **ymax** définissent la zone de dessin : entre **-xmax** et **+xmax** en abscisse et **-ymax** et **ymax** en ordonnées

– (illusion de) retourner plusieurs valeurs –

- Comme une fonction peut retourner un **tuple**, on peut s'en servir pour retourner plusieurs valeurs

```
def divEuclidienne(a,b):
    'retourne quotient et reste'
    return a // b, a % b

q,r = divEuclidienne(14,4)
print('quotient=',q,', reste=',r)
```

Python pas à pas

Lecture / écriture dans un fichier

– L'instruction join( ) –

- l'instruction **join( )** est une instruction (méthode) de chaîne de caractères
- On l'utilise de la façon suivante :

  s.join(it)

  où **s** est une chaîne de caractères et **it** est un itérable contenant des chaînes de caractères.
- le résultat est une chaîne qui contient les mots de **it** reliés par **s**
- **':'.join(['ab','cd','efg']** $\rightarrow$ **'ab:cd:efg'**

– Ouverture et fermeture d'un fichier –

- On peut instancier une variable de type **fichier**, qui va permettre de faire des opérations sur les fichiers présents sur l'ordinateur
- Pour **ouvrir** un fichier en python, on utilise la commande :

  f = open(chemin,mode),

  où **f** est la variable qu'on utilisera pour accéder au fichier, **chemin** est le nom du fichier (éventuellement avec le chemin pour le trouver **'../toto.txt'**) et **mode** est le mode d'utilisation du fichier dans le programme
- Il existe de nombreux modes d'accès aux fichiers, voilà les trois plus communs :
  - **'r'** : mode lecture seulement, c'est le mode par défaut
  - **'w'** : mode écriture, le fichier est créé s'il n'existe pas, sinon il est effacé pour pouvoir y écrire
  - **'a'** : mode ajout, c'est un mode écriture à partir de la fin du fichier
- Pour **fermer** un fichier : **f.close()**

– Les objets file –

- C'est un objet **modifiable** : si on le fait évoluer dans une fonction, il évolue globalement
- Il connaît le fichier
- Il a une **position courante** dans le fichier, qui est modifiée au fur et à mesure qu'on lit ou écrit dans le fichier

– Lecture dans un fichier –

- Il faut que le fichier soit **ouvert en lecture**
- On peut lire une ligne de **f** avec l'instruction **f.readline()**
- Cela **déplace la position courante** à la ligne suivante
- Donc on peut répéter l'appel à **f.readline()** pour lire toutes les lignes une à une
- Quand il n'y a plus rien à lire, **f.readline()** retourne la chaîne vide ''

```
f = open('filtre.py')
ligne = None
while ligne != '':
    ligne = f.readline()
    print(l)
f.close()
```

– Solution alternative –

- Une file **f** peut aussi être vue comme une **structure itérable** de ses lignes
- Cela permet de très facilement lire les lignes de **f**

```
f = open('iterable.py')
for ligne in f:
    print(ligne)
f.close()
```

Que l'on utilise **readline()** où le format d'itérable, les lignes retournées conservent le caractère **'\ n'** à la fin. On peut l'enlever avec l'instruction **ligne = ligne[:-1]** (cf dernier cours)

– Ecriture –

- Pour écrire dans un fichier, il faut l'ouvrir en **écriture 'w'** ou en **ajout 'a'**
- Pour écrire la chaîne **s** dans le fichier **f**, on utilise l'instruction **f.write(s)**
- Attention, contrairement à **print( )**, cela ne rajoute pas un saut de ligne à la fin

```
f = open('tmp','w')
for i in range(10):
    f.write('ligne '+str(i)+'\n')
f.close()
```

– Exemples –

- Lister les palindromes en français
- Créer un nouveau fichier sans les accents et sans les ç
- Lister les palindromes du nouveau fichier
- Le jeu du pendu
- Recherche d'anagrammes :
  - Un ensemble de lettres (avec répétitions) est vu comme un tuple ordonné (on utilise la fonction **lst.sort()** qui tri la liste **lst**
  - On stocke les anagrammes sous forme d'un dictionnaire où les clés sont les tuples ordonnés ci-dessus, et les valeurs l'ensemble des mots qui utilisent ces lettres

Introduction à l'algorithmique et à la programmation

IUT 1ère année 2013-2014

Cyril Nicaud

Cyril.Nicaud@univ-mlv.fr

– Cours 5 / 5 –

Python pas à pas

Les slices

– Notion de slice –

- On a vu qu'on peut acceder au *i*-ème élément d'une liste ou d'une chaîne avec **t[i]**
- Le **slice** consiste à accéder à une **portion** d'une liste ou d'une chaîne
- **Notation : t[debut:fin]** prend la sous-liste où la sous-chaîne comprise entre les indices **debut** et **fin-1**
- **Attention :** c'est **fin -1** comme pour les **range**

s = 'bonjour'
print(s[2:5]) $\rightarrow$ **'njo'**

– Indices négatifs –

- On peut utiliser des **indices négatifs**
- L'indice **-i** est le même que **len - i**

0	1	2	3	4	5	6
b	**o**	**n**	**j**	**o**	**u**	**r**
-7	-6	-5	-4	-3	-2	-1

s = 'bonjour'
print(s[-2]) $\rightarrow$ **u**
print(s[1:-2]) $\rightarrow$ **onjo**

– Paramètres par défaut –

- Dans un **slice**, le début est par défaut 0 : **t[:4]** est la même chose que **t[0:4]**
- Dans un **slice**, la fin est par défaut **len(t)** : **t[4:]** est la même chose que **t[4:len(t)]**

- **t[:7]** ce sont donc les 7 premiers éléments
- **t[2:]** ce sont les éléments à partir du troisième (le premier est à 0
- **t[:-2]** ce sont tous les éléments sauf les 2 derniers
- **t[-5:]** ce sont les 5 derniers éléments

Lors d'un **slice**, Python recopie la portion de chaîne (ou de liste, ...) qui est extraite.

Python pas à pas

Notion d’exception

– Qu'est-ce qu'une exception ? –

- Quand un programme plante, c'est qu'il y a eu un problème qui a **levé une exception**
- Le **mécanisme d'exception** sert à signaler une **anomalie** de fonctionnement
- Quand une telle **anomalie** se produit, on peut dans le programme :
  - ne rien faire et laisser le programme planter
  - **intercepter** l'exception et traiter le problème dans le programme

Message d'erreur type :

Traceback (most recent call last):
  File "code4/erreur.py", line 1, in <module>
    x = 3 // 0
ZeroDivisionError: integer division or modulo by zero

– Mécanisme d'interception –

- Pour **intercepter une exception**, il faut mettre le code qui risque d'en générer une dans un bloc **try:**
- L'interception se fait ensuite dans un bloc **except:**

```
try:
    instructions à risque
except:
    instructions en cas d'exception
```

– Exemple : saisir un entier –

- la fonction demande un nombre et tente de le convertir en entier avec la fonction **int( )**
- si elle n'y arrive pas, une exception est levée, qui est interceptée avec le **except**
- en cas de problème on retourne **None** : le programme ne plante pas, on peut redemander le nombre

```
def saisieNombre(s='nombre = '):
    try:
        return int(input(s))
    except:
        return None
```

– Interception (suite) –

- Un même code peut générer **plusieurs types d'erreurs**
- Il peut être utile de savoir les **distinguer** lors de l'interception.

```
try:
    n = int(input('nombre = '))
    print(1 / n)
except:
    print('il y a eu une erreur')
```

– Noms d'exceptions –

- On peut paramétrer les **except** avec un nom d'exception
- Si un **except** est paramétré, il n'est exécuté que si une exception **du bon nom** est levée
- Le **nom** de l'exception est celui indiqué sur la dernière ligne du message d'erreur

```
Traceback (most recent call last):
  File "code4/erreur.py", line 1, in <module>
    x = 3 // 0
ZeroDivisionError: integer division or modulo by zero
```

– Exemple avec plusieurs except –

```
try:
    n = int(input('nombre = '))
    print(1 / n)
except ZeroDivisionError:
    print('Division par zero!')
except:
    print('il y a eu une erreur')
```

– Lever sa propre exception –

- Il est possible de **lever** volontairement une exception pour signaler un problème
- L'instruction est **raise NameError(str)**, où **str** est un message d'information
- **Attention :** le **nom** d'une telle exception est **NameError**

```
def puissance(x,n):
    if n < 0:
        raise NameError('pas de puissance negative')
    r = 1
    for i in range(n):
        r *= x
    return r
```

Python pas à pas

Les listes en compréhension

– Listes en compréhension –

- L'idée est d'avoir un moyen de décrire une liste comme on décrit un **ensemble en mathématiques** :

$$E = \{2^i \mid i \in \{0 \cdots 10\}\}$$
$$F = \{x \in \{10 \cdots 30\} \mid x \text{ impair}\}$$

- Pour le premier, la syntaxe est **[f(x) for x in iterable]**, où **f** est une fonction :

```
E = [2**i for i in range(11)]
```

- pour ajouter une condition, on utilise **if** :

```
F = [i for i in range(11) if i%2 == 1]
```

– Liste en compréhension (suite) –

- On peut utiliser plusieurs **for** :

  C = [(i,j) for i in range(5) for j in range(4)]

- On peut bien entendu s'en servir sur d'autres types que les **int**

```
s = "Ceci est le dernier cours de Python"
lst = [(u,len(u)) for u in s.split()]
for x in lst:
    print(x)
lst2 = [u for u in s.split() if 'e' not in u]
print(lst2)
```

– Quelques exemples –

- Les premières lettres de chaque mot d'une phrase
- Les nombres premiers via les nombres non-premiers
- Une application aléatoire de $\{1, ..., n\}$ dans $\{1, ..., n\}$
- Les racines de $x^5 - 5x^3 + 4x$
- Les entêtes de fonctions dans un fichier Python

– Fonctions anonymes : fonctions lambda –

- Il peut être utile de créer une fonction à la volée pour la passer en **paramètre**
- On peut le faire grâce au mot clé **lambda**, la syntaxe est

lambda x : expression(x)

qui est une fonction anonyme qui est l'équivalent de
def f(x):
    return expression(x)

```
s = "ceci est le dernier cours de python"
lst = s.split()
print(sorted(lst,key=lambda x:x[0]))
```