Cours d'algorithmique et programmation 1

Tiago de Lima

UNIVERSITÉ D'ARTOIS
Faculté des Sciences Jean Perrin
Licences Mathématiques et Informatique

Année universitaire 2019–2020

Section 1

Présentation de l'unité

Objectif de l'unité

L'**objectif** de cette unité est de faire acquérir aux étudiants des notions et concepts de base de la programmation impérative.

Seront abordées :

- les notions basiques ;
- les instructions conditionnelles ;
- les instructions itératives ;
- les fonctions ;
- les types composites :
  - listes, dictionnaires.

Contenu de l'unité

ECTS : 6

Cours magistraux (CM) : $12 \times 1{,}5$ h = 18 h

Travaux dirigés (TD) : 12×2 h = 24 h

Travaux pratiques (TP) : 12×2 h = 24 h

Plan (intention) :

Semaine	CM	TD	TP
1	Expressions	—	—
2	Entrées et sorties		
3	Conditions		
4	Itérations		
5	Fonctions et modules (1)		
6	Fonctions et modules (2)		
7	Types composites		
Int.	**Devoir surveillé**	—	—
8	Graphismes		
9	Animation		Projet
10	Jeux		Projet
11	Images, son et contrôle		Projet
12	Détection de collisions		Projet
13	—		**Dépôt du projet**

Modalités de contrôle de connaissance

- Session 1 :
  - **3 TP notés :** tirés au hasard (semaines 2–8)
  - **Devoir surveillé :** octobre 2019 (semaine d'interruption)
  - **Projet :** semaines 9–13 (dépôt vers le 14 décembre)
  - **Examen :** décembre 2019 (semaine d'examens)

$$Note = \frac{(\frac{TP+DS}{2} + EX_1 + P)}{3}$$

- Session 2 :
  - **Examen de rattrapage :** en juin 2020

$$Note\ Finale = \max(Note, \frac{Note + EX_2}{2})$$

Site web de l'unité

Moodle : **`http://moodle.univ-artois.fr/`**

Clé : `algo1-1920`

Vous trouverez :

- supports des cours (diaporama)
- liens
- annonces
- exercices
- dépôt du projet
- notes

Lecture supplémentaire

Télécharger Thonny : **`https://thonny.org`**

Site officel Python : **`http://python.org`**

Site officiel Pygame : **`http://pygame.org`**

Livre en français : `http://inforef.be/swi/python.htm`

Apprendre à programmer avec Python 3
Gérard Swinnen. Eyrolles, 3e édition. Chapitres 1 à 7

Cours en ligne (en anglais) : `http://programarcadegames.com`

Program Arcade Games with Python and Pygame
Paul Vincent Craven.

Livre en anglais : `http://inventwithpython.com`

Invent Your Own Computer Games with Python
Albert Sweigart. 2e édition.

Cours 1

Expressions

Sommaire

Section 2

Quelques notions fondamentales

Informatique

L'**informatique** est le domaine d'activité scientifique, technique et industriel concernant le traitement automatique de l'information via l'exécution des programmes informatiques par des machines : des systèmes embarqués, des ordinateurs, des robots, des automates, etc.

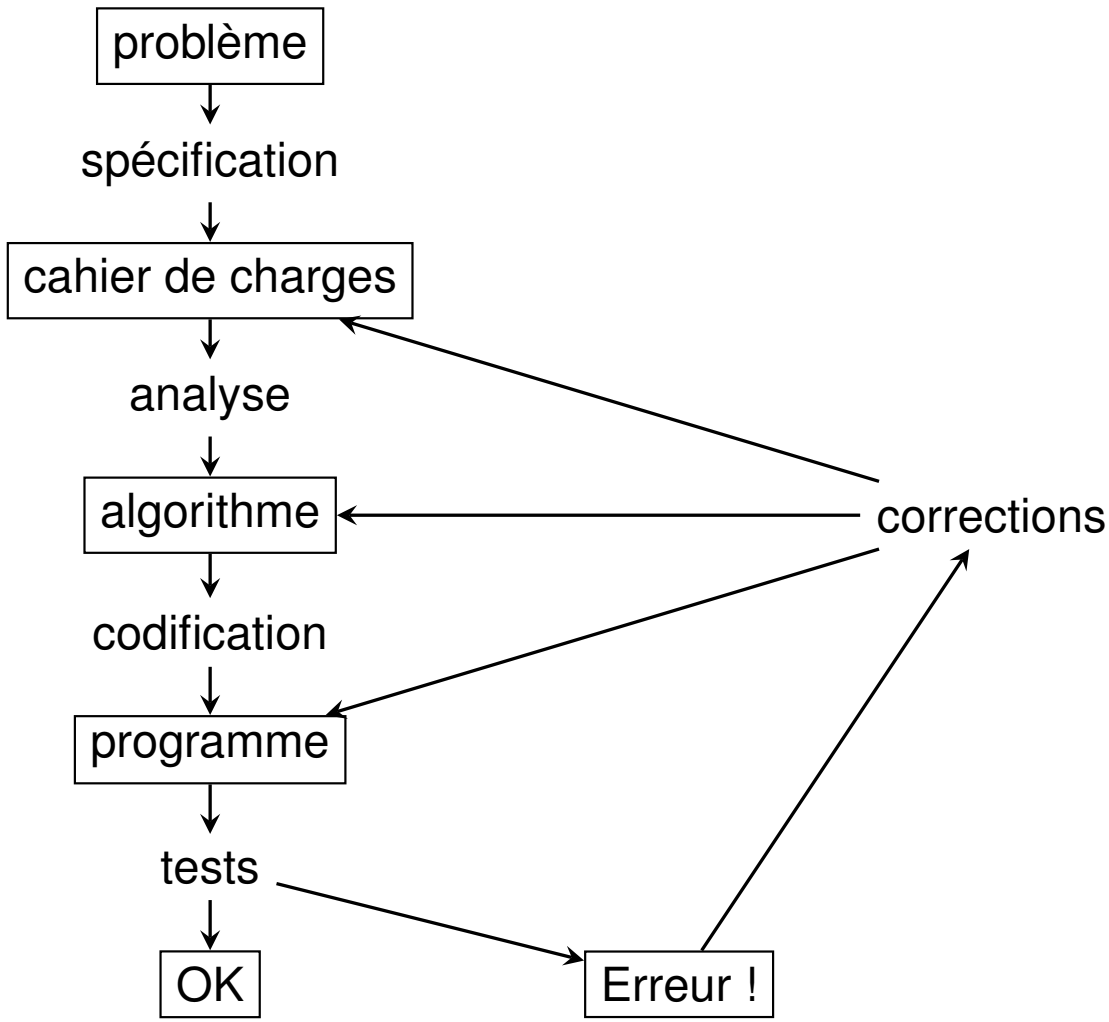
– `http://fr.wikipedia.org/wiki/informatique`

Dans cette unité, nous nous intéressons à la programmation des ces machines.

Programmation

La **programmation** est l'élaboration et codification du traitement à effectuer pour résoudre un problème.

Les étapes de la programmation :



Algorithme

Un **algorithme** est un ensemble d'opérations élémentaires, organisé dans le but de résoudre un problème donné.

Exemples de problèmes :

- préparer une tarte aux fraises ;
- trouver un mot dans le dictionnaire ;
- résoudre une équation.

Exemples d'opérations élémentaires :

- mélanger les ingrédients ;
- comparer deux mots ;
- faire l'addition de deux entiers.

Programme et langage de programmation

Un **programme** est un algorithme écrit dans un langage de programmation.

Un **langage** est un ensemble de phrases.

Exemples de phrases :

- en français (« Je m'appelle Paul. ») ;
- en anglais (« The book is on the table ! ») ;
- en arithmétique (« $1 + 2 = 3$ ») ;

Un **langage de programmation** est un langage qui sert a codifier des algorithmes afin de les rendre « compréhensibles » par la machine.

Exemple en Python :

```
def succ(x=0):
    return x + 1
```

Différents langages de programmation

Il existe plusieurs langages de programmation :

- impératifs (C, Pascal, FORTRAN, . . .) ;
- fonctionnels (Lisp, Haskel, Scheme, ML, . . .) ;
- orientés objet (SmallTalk, C++, Java, . . .) ;
- déclaratif (PROLOG).

Notre langage de programmation

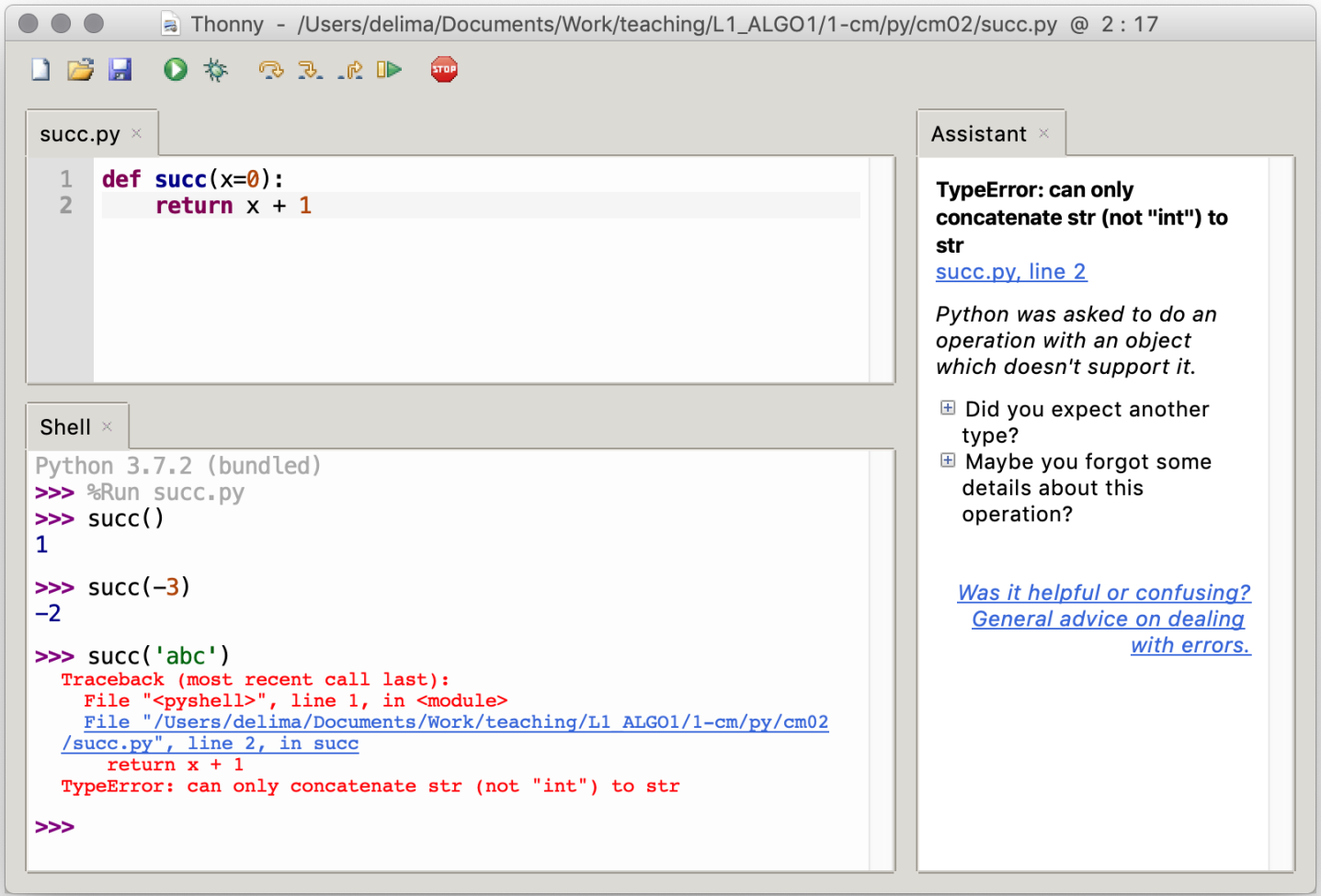
Nous apprendrons à programmer en utilisant le langage **Python**.

Pourquoi ?

- Gratuit
- Syntaxe simple
- Portable (Mac OS, Unix, Windows, Android, etc.)
- Compilé et interprété avec une machine virtuelle
- Multi-paradigme (impératif, orienté objet et fonctionnel)
- Moderne (dynamiquement typé, multi-threadé, extensible, etc.)
- Utilisé dans des nombreux projets industriels :
  - Arts, business, éducation, ingénierie, gouvernement, scientifique, etc.
  - Regarder : `https://www.python.org/about/success/`
- En évolution

Thonny

Pour nos cours et TP, nous utiliserons Thonny.



Section 3

Python shell

Python shell

Le mode interactif de Python est aussi appelée **Python shell.**

Par exemple, nous pouvons lui demander de faire une petite somme :

```
>>> 2 + 2
4
```

Notre instruction est une expression arithmétique qui renvoie un résultat. Python évalue et imprime le résultat de cette expression : `4`.

Attention : **l'invite, c'est-à-dire, les caractères** >>> **ne font pas partie de l'instruction**.

L'invite indique tout simplement que Python attend une instruction.

Dans l'exemple, l'instruction donnée à Python est uniquement :

```
2 + 2
```

Section 4

Expressions

Expressions

Une **expression** en Python est formée des valeurs, des opérateurs et des parenthèses.

Les règles qui définissent les expressions valables en Python sont pratiquement les mêmes que pour les expressions arithmétiques.

Par exemple, ceci est une expression valable en Python :

```
(10 - 5)/ 2
```

Quand elle est donnée à Python, ce dernier l'évalue et imprime le résultat :

```
>>> (10 - 5) / 2
2.5
```

Expressions

Nous pouvons écrire des expressions avec et sans espaces entre les opérateurs et les valeurs :

```
>>> 2+2
4
>>> 10    -5+    6
11
```

Valeurs numériques

Les nombres comme `10` et `-5` sont appelés nombres **entiers** (`int`).

```
>>> 10
10
>>> -5
-5
```

En Python, le nombre `5` est un int, mais le nombre `5.0` ne l'est pas.
Il est un nombre à **virgule flottante** (`float`).

```
>>> 10.5
10.5
>>> -1.5
-1.5
>>> 14.5e30
1.45e+31
```

Attention : dans tous les langages de programmation (donc, en Python aussi) **le séparateur décimal est un point**, et non une virgule.

Opérateurs arithmétiques

Python a des nombreux opérateurs pour les int et les float :

opération	**opérateur**	**exemple**
addition	`+`	`3 + 2`
soustraction	`-`	`3 - 2`
multiplication	`*`	`3 * 2`
division	`/`	`10 / 3`
division entière	`//`	`10 // 3`
exponentiation	`**`	`2 ** 3`
modulo	`%`	`8 % 3`

Exemples :

```
>>> 10 / 3
3.3333333333333335
>>> 10 // 3
3
>>> 8 % 3
2
```

Ordre de priorité des opérateurs

Python évalue les opérateurs comme en arithmétique.

Par exemple, l'expression :

```
90 + 86 + 71 + 100 + 98 / 5
```

est équivalente à :

$$90 + 86 + 71 + 100 + \frac{98}{5}$$

et l'expression :

```
(90 + 86 + 71 + 100 + 98) / 5
```

est équivalente à :

$$\frac{90 + 86 + 71 + 100 + 98}{5}$$

Exercice 1

Problème :

Calculer la consommation de notre voiture aux 100 km, sachant que nous avons parcouru 679 km avec 31 litres de carburant.

Algorithme :

$$100 \times \frac{31}{679}$$

Programme :

```
100 * (31 / 679)
```

Test :

```
>>> 100 * (31 / 679)
4.565537555228277
```

Exercice 2

Problème :

Nous voulons savoir combien notre voyage allez-retour en voiture de Lens jusqu'à Rennes nous coûtera. La distance entre les deux villes est de 613 km. Nous venons de calculer la consommation de la voiture. L'essence est à 1,62 € le litre et le péage coûtera 32,40 €.

Algorithme :

$$(4.57 \times \frac{613}{100} \times 1.62 + 32.40) \times 2$$

Programme :

```
(4.57 * (613 / 100) * 1.62 + 32.4) * 2
```

Test :

```
>>> (4.57 * (613 / 100) * 1.62 + 32.4) * 2
155.56568400000003
```

Section 5

Variables

Variables et noms de variables

Une **variable** est une position de mémoire où une valeur est enregistrée.

En Python, un **nom de variable** est une suite de caractères commençant par une lettre ou le symbole « _ » et pouvant comporter :

- des lettres
- des chiffres
- le symbole « _ »

Noms de variables

Le guide de style de Python recommande le **snake case :**

- les noms de **variables** doivent être écrits en lettres minuscules
- les noms de **constantes** doivent être écrit en majuscules
- les noms comptant plus d'un mot doivent s'utiliser du symbole « _ » pour les séparer

Exemples :

permis	**non-recommendés**	**interdits**
`vitesse`	`Vitesse`	`1vitesse`
`vitesse_max`	`vitesseMax`	`vitesse max`
`vitesse_voiture`	`VitesseVoiture`	`vitesse-voiture`
`VITESSE_MAX`	`VITESSE_max`	`VITESSE-MAX`

Section 6

Instruction d'affectation

Instruction d'affectation

L'**instruction d'affectation** sert à enregistrer une valeur dans une variable.

Syntaxe :

$$\boxed{var = expr}$$

Exemples :

```
>>> x = 10
>>> x
10
>>> x + 1
11
>>> y = x + 1
>>> y
11
>>> x + y
21
```

Instruction d'affectation

Attention : en Python, le symbole « = » ne fonctionne pas du tout comme en algèbre.

```
>>> x = 1
>>> x
1
>>> x = x + 1
>>> x
2
```

Attention : l'affectation est toujours faite aux variables.

```
>>> x + 1 = x
File "<stdin>", line 1
    SyntaxError: can't assign to operator
```

Erreur de syntaxe : impossible d'affecter à un opérateur.
(`x + 1` n'est pas un nom de variable.)

Utilisation de variables

Nous allons refaire l'exercice du calcul du coût du voyage (Exercice 2) en utilisant des variables.

1. D'abord, nous enregistrons les valeurs du problème dans des variables :
   - la voiture fait 4,57 litres au 100 km.
   - la distance entre les deux villes est de 613 km.
   - le prix du carburant est de 1,62 € le litre.
   - le péage coûtera 32,40 €.

```
>>> conso = 4.57
>>> dist = 613
>>> prix_carb = 1.62
>>> peage = 32.40
```

Utilisation de variables

2. Ensuite, nous calculons le coût du carburant pour un aller simple en utilisant les variables appropriées :

```
>>> cout_carb = conso * (dist / 100) * prix_carb
```

3. Nous calculons le coût total du voyage aller-retour :

```
>>> total = (cout_carb + peage) * 2
```

4. Enfin, nous affichons le résultat :

```
>>> total
155.56568400000003
```

Section 7

Utilisation d'un fichier source

Utilisation d'un fichier source

Le programme du calcul du coût du voyage (Exercice 2) contient plusieurs instructions.

Nous allons maintenant créer un programme un peu plus long.

Pour des programmes comme cela, il est plus convenable d'utiliser un fichier.

Nous allons enregistrer une suite d'instructions dans un fichier texte.

Ensuite, nous demanderons à Python d'exécuter les instructions du fichier enregistré.

Utilisation d'un fichier source

Voici à quoi ressemble notre programme enregistré dans un fichier :

cout_voyage_1.py

```
1 conso = 4.57
2 dist = 613
3 prix_carb = 1.62
4 peage = 32.40
5 cout_carb = conso * (dist / 100) * prix_carb
6 total = (cout_carb + peage) * 2
```

```
>>> %Run cout_voyage_1.py
>>> total
155.56568400000003
```

Après son exécution, le programme enregistre le résultat dans la variable `total`.

Pour l'afficher, il suffit de demander sa valeur à Python.

Paramétrage d'un programme

Pourtant, le programme `cout_voyage_1.py` fonctionne uniquement pour les valeurs données.

Pour calculer le coût d'un autre voyage, nous devons soit :

- changer les valeurs dans le programme et le relancer ; ou bien
- écrire un autre programme.

Idéalement, nous voudrions écrire un programme plus général, où les valeurs sont données par l'utilisateur.

En effet, nous voulons « paramétrer » le programme.

Section 8

Fonctions

Exemples de fonction

Une manière de paramétrer notre programme est utiliser une **fonction**.

Exemple : Définition d'une fonction qui calcule la valeur 1.

fonc_1.py

```
1 def fonc_1():
2     x = 0
3     return x + 1
```

```
>>> %Run fonc_1.py
>>> fonc_1()
1
```

Le mot-clé `def` sert à « définir » une fonction.

Le mot-clé `return` sert à « retourner » la valeur calculée par la fonction.

Pour exécuter le programme, nous choisissons l'option « Run current script » de Thonny.

Pour appeler la dans le shell, nous écrivons son nom avec les parenthèses.

Exemples de fonction

Voici un exemple plus intéressant.

Exemple : Définition d'une fonction qui calcule le successeur d'un nombre.

succ.py

```
1 def succ(x=0):
2     return x + 1
```

```
>>> %Run succ.py
>>> succ()
1
>>> succ(1)
2
>>> succ(2)
3
```

Pour appeler cette fonction, nous écrivons son nom avec les arguments entre parenthèses.

Exemples de fonction

Dans l'exemple précédent, la variable `x` entre parenthèses est un **paramètre** (aussi appelée **argument**) de la fonction.

L'argument `x` est initialement définit avec la valeur `0`, mais il est possible de le changer.

L'instruction `succ(1)` fait appel à la fonction `succ` en donnant la valeur `1` à la variable `x`.

L'instruction `succ(2)` fait le même avec `2`.

Il est aussi possible d'utiliser la syntaxe suivante également :

```
>>> succ(x=1)
2
>>> succ(x=2)
3
```

Ceci s'avère très pratique dans l'exemple suivant.

Exemples de fonction

Exemple : Une fonction qui calcule la somme de deux nombres.

somme.py

```
1 def somme(x=0, y=0):
2     return x + y
```

```
>>> %Run somme.py
>>> somme()
0
>>> somme(1)
1
>>> somme(1, 1)
2
>>> somme(1, y=1)
2
>>> somme(x=3, y=-2)
1
>>> somme(y=2, x=2)
4
```

Exemples de fonction

Pourtant, tout n'est pas permis :

```
>>> somme(x=1, 2)
File "<pyshell>", line 1
    SyntaxError: positional argument follows keyword argument

>>> somme(x=1, x=2)
File "<pyshell>", line 1
    SyntaxError: keyword argument repeated

>>> somme(1, x=2)
Traceback (most recent call last):
    File "<pyshell>", line 1, in <module>
    TypeError: somme() got multiple values for argument 'x'
```

Définition de fonction

Les **fonctions** servent à deux choses :

- rendre le programme plus lisible et organisé
- permettre la réutilisation du code

Syntaxe de la définition d'une fonction :

```
def fonc([var_1[=expr_1], ..., var_n[=expr_n]]):
    suite
```

`[ ]` : les éléments dans des crochets sont optionnels

`fonc` : nom de la fonction

`var_i` : variable qui stockera une valeur passée en argument

`expr_i` : valeur par défaut stockée dans la variable *`var_i`*

`suite` : une suite d'instructions

Appel de fonction

Syntaxe de l'appel (exécution) d'une fonction :

```
fonc(arg_1, ..., arg_n)
```

fonc: nom de la fonction

arg_i: un argument passé à la fonction

Exercice

Problème :

Écrire une programme de calcul du coût du voyage. Le programme doit être écrit avec une fonction parametrée par les informations suivantes :

- la consomation au 100 km ;
- la distance à parcourir ;
- le prix du carburant ;
- le coût du péage.

Exercice

Programme :

cout_voyage_2.py

```
1 def calc_cout(conso=4.57,
2                dist=613,
3                prix_carb=1.62,
4                peage=32.4):
5     cout_carb = conso * (dist / 100) * prix_carb
6     total = (cout_carb + peage) * 2
7     return total
```

Exercice

Tests :

```
>>> %Run cout_voyage_2.py
>>> calc_cout()
155.56568400000003

>>> calc_cout(dist=40, peage=0)
5.922720000000002

>>> calc_cout(dist=40, peage=0, prix_carb=1.4)
5.1184

>>> calc_cout(dist=40, peage=0, prix_carb=1.4, conso=3.2)
3.5840000000000005
```

Les fonctions dans des expressions

Les fonctions peuvent être utilisées dans des expressions.

```
>>> %Run succ.py
>>> succ(2) + 3
6
>>> x = succ(2) + 3
>>> x + 1
7
```

Ceci veut dire que nous pouvons faire des calculs avec le programme du coût d'un voyage aussi.

Par exemple, pour calculer le total de deux voyages :

```
>>> %Run cout_voyage_2.py
>>> calc_cout(dist=40, peage=0) + calc_cout(dist=80,
    peage=0)
17.768160000000005
```

Cours 2

Entrées et sorties

Précédemment sur ALGO1

Nous avons vu :

- Quelques notion fondamentales, notamment :

  Un **algorithme** est un ensemble d'opérations élémentaires, organisé dans le but de résoudre un problème donné.

- Python shell :

```
>>> 2 + 2
4
```

- Instruction d'affectation :

```
>>> x = 1
>>> x = x + 1
>>> x
2
```

Précédemment sur ALGO1

- Définition et appel des fonctions :

succ.py

```
1 def succ(x=0):
2     return x + 1
```

```
>>> %Run succ.py
>>> succ(x=2)
3
```

Précédemment sur ALGO1

- Calcul du coût d'un voyage en voiture :

```
>>> %Run cout_voyage_1.py
>>> cout_du_voyage()
155.56568400000003

>>> cout_du_voyage(dist=40, peage=0)
5.922720000000002

>>> cout_du_voyage(dist=40, peage=0, prix_carb=1.4)
5.1184

>>> cout_du_voyage(dist=40, peage=0, prix_carb=1.4,
    conso=3.2)
3.5840000000000005
```

Aujourd'hui

Nous allons améliorer le programme de calcul du coût d'un voyage en voiture.

Notre programme va interagir avec l'utilisateur.

Exemple d'exécution :

```
Calcul du coût d'un voyage en voiture !
---------------------------------------
Consommation au 100 km : 4.57
Distance à parcourir : 45
Prix du carburant : 1.62
Coût du péage : 0

Coût total du voyage : 6.663060000000015
```

Sommaire

Section 9

Entrées de données

Entrées de données

Actuellement, notre programme est général. Il peut calculer le coût de tous les voyages en voitures entre deux villes.

Pourtant, l'utilisateur doit se souvenir des noms des variables utilisées dans le code (`conso`, `dist`, etc.).

Nous allons améliorer ce point. Nous allons faire le programme demander chaque valeur à l'utilisateur.

Fonction input

La fonction **input** permet à l'utilisateur d'interagir avec un programme en Python. Elle permet à l'utilisateur d'entrer des données.

Syntaxe de la fonction input :

```
input(prompt) -> str
```

prompt: message affiché à l'utilisateur

`-> str:` ceci veut dire que la fonction retourne une **chaîne de caractères** (`str`).

Exemple :

```
>>> ch = input('Entrez une donnée : ')
Entrez une donnée : test
>>> ch
'test'
```

Fonction input

Attention : la fonction `input` renvoie toujours une chaîne de caractères.

Une str n'est pas un nombre (comme les int ou float).

Donc, nous ne pouvons pas traiter l'entrée renvoyée par input comme un nombre.

```
>>> x = input('Entrez un nombre : ')
Entrez un nombre : 10
>>> x + 1
Traceback (most recent call last):
    File "<pyshell>", line 1, in <module>
    TypeError: can only concatenate str (not "int") to str
```

Erreur de type.

Section 10

Conversions de valeurs

Conversions de valeurs

Sous certaines conditions, il est possible d'effectuer la conversion d'une chaîne de caractères en nombre avec les fonctions `int` et `float`.

Syntaxe de la fonction int :

```
int(expr) -> int
```

Syntaxe de la fonction float :

```
float(expr) -> float
```

La fonction `int` reçoit une expression `expr` de type str ou float comme argument et revoie l'int correspondant.

La fonction `float` reçoit une expression `expr` de type str ou int comme argument et renvoie le float correspondant.

Conversions de valeurs

Exemples :

```
>>> ch1 = input('Entrez un entier : ')
Entrez un entier : 10
>>> x = int(ch1)
>>> x + 1
11
>>> ch2 = input('Entrez un flottant : ')
Entrez un flottant : 5.5
>>> y = float(ch2)
>>> y + 3.3
8.8
```

Ou bien, de manière plus compacte :

```
>>> x = int(input('Entrez un entier : '))
Entrez un entier : 10
>>> x + 1
11
>>> y = float(input('Entrer un flottant : '))
Entrez un flottant : 5.5
>>> y + 3.3
8.8
```

Conversions de valeurs

Mais attention aux possibles problèmes :

```
>>> ch3 = input('Entrez un nombre : ')
Entrez un nombre : texte
>>> z = int(ch3)
Traceback (most recent call last):
    File "<pyshell>", line 1, in <module>
    ValueError: invalid literal for int() with base 10: 'texte'

>>> ch4 = input('Entrez un nombre : ')
Entrez un nombre : 10.5
>>> w = int(ch4)
Traceback (most recent call last):
    File "<pyshell>", line 1, in <module>
    ValueError: invalid literal for int() with base 10: '10.5'
```

Exercice

Problème :

Compléter le programme de calcul du coût du voyage de manière à ce que le programme demande à l'utilisateur les données suivantes :

- la consommation au 100 km ;
- la distance à parcourir ;
- le prix du carburant ;
- le coût du péage.

Ensuite, le programme doit calculer le coût du voyage.

Exercice

Programme :

cout_voyage_3.py

```
1  def calc_cout(conso=4.57, dist=613, prix_carb=1.62,
       peage=32.4):
2      cout_carb = conso * (dist / 100) * prix_carb
3      total = (cout_carb + peage) * 2
4      return total
5
6  def main():
7      w = float(input('Consommation au 100 km : '))
8      x = float(input('Distance à parcourir : '))
9      y = float(input('Prix du carburant : '))
10     z = float(input('Coût du péage : '))
11     result = calc_cout(w, x, y, z)
12     return result
```

Exercice

Test :

```
>>> %Run cout_voyage_3.py
>>> main()
Consommation au 100 km : 4.57
Distance à parcourir : 613
Prix du carburant : 1.62
Coût du péage : 32.4
155.56568400000003
```

Section 11

Affichage à l'écran

Affichage à l'écran

Notre programme du coût d'un voyage est presque fini.

Il nous reste pouvoir afficher un joli message au début. Par exemple :

```
Calcul du coût d'un voyage en voiture !
---------------------------------------
```

Et à la fin. Par exemple :

```
Coût total du voyage : 6.6630600000000015
```

Fonction print

Syntaxe de la fonction print :

`print([expr_1, ..., expr_n])`

expr_i: une expression qui sera affichée à l'écran

Exemple :

```
>>> peage = 32.4
>>> print(peage)
32.4
>>> print('Coût du péage :')
Coût du péage
```

Affichage de résultats d'expressions

Il est possible d'afficher le résultat d'une expression :

```
>>> print(2 + 2)
4
```

Il est possible d'afficher plusieurs expressions avec un seul appel à la fonction `print`. Il suffit d'utiliser une petite virgule :

```
>>> print(1.45 * 19, 1030 + 10)
27.55 1040
>>> peage = 32.4
>>> print('Coût du péage :', peage)
Coût du péage : 32.4
```

Section 12

Chaînes de caractères

Chaînes de caractères

Le type **chaîne de caractères** (`str`) représente les textes en Python.

Exemples :

```
>>> print('exemple')
exemple
>>> ch = 'Ceci est un exemple.'
>>> print(ch)
Ceci est un exemple.
```

Notez la présence (obligatoire !) des guillemets anglais simples.

Les guillemets anglais doubles peuvent être utilisés aussi :

```
>>> print("exemple")
exemple
>>> ch = "Ceci est un exemple."
>>> print(ch)
Ceci est un exemple.
```

Chaînes de plusieurs lignes

Les chaînes de caractères de plusieurs lignes peuvent être insérées en utilisant trois guillemets simples ou doubles pour les délimiter :

```
1 print('''Ceci est une chaîne
2 de plusieurs lignes.
3 Python va l'afficher.''')
4 print("""Ceci est une
5 autre chaîne
6 de plusieurs lignes.""")
```

```
>>> %Run ...
Ceci est une chaîne
de plusieurs lignes.
Python va l'afficher.
Ceci est une
autre chaîne
de plusieurs lignes.
```

Séquences d'échappement

```
>>> print('Python, c'est facile !')
File "<stdin>", line 1
    print('Python, c'est facile !')
                      ^
    SyntaxError: invalid syntax
```

Pour construire de chaînes de caractères contenant des guillemets simples, nous pouvons utiliser des guillemets doubles et vice-versa :

```
>>> print("Python, c'est facile !")
Python, c'est facile !
```

Ou bien, une **séquence d'échappement** :

```
>>> print('Python, c\'est facile !')
Python, c'est facile !
>>> print('J\'ai dis \"Oui !\"')
J'ai dis "Oui !"
```

Séquences d'échappement

Quelques séquences d'échappement utiles :

séquence	**affichage**
`\"`	guillemets doubles
`\'`	guillemets simples
`\\`	anti-slash
`\t`	tabulation
`\n`	nouvelle ligne

Exemple :

```
>>> print("Ceci\nest\nun\nexemple.")
Ceci
est
un
exemple.
```

Opérateurs de chaînes de caractères

Les chaînes de caractères sont aussi des valeurs en Python.

Nous pouvons donc créer des expressions avec elles.

Python a quelques opérateurs pour les chaînes de caractères :

opérateur	**opération**	**exemple**
`+`	concaténation	`'Bonjour ' + 'monsieur !'`
`*`	répétition	`3 * 'Bonjour ! '`
`[ ]`	indexation	`'Bonjour'[0]`

Opérateurs de chaînes de caractères

Exemples :

```
>>> ch1 = 'Bonjour'
>>> ch2 = ' madame !'
>>> print(ch1 + ch2)
Bonjour madame !
>>> print(3 * (ch1 + ' ! '))
Bonjour ! Bonjour ! Bonjour !
>>> print(ch2[0] + ch2[1] + ch2[2])
 ma
>>> ch = ch1[-4] + ch1[-3] +  ch1[-2] + ch1[-1]
>>> print(ch)
jour
```

Exercice

Problème :

Ajouter au programme du coût d'un voyage, les messages au début et à la fin du programme.

Exercice

Programme :

cout_voyage_4.py

```
1  def calc_cout(conso=4.57, dist=613, prix_carb=1.62,
       peage=32.4):
2      ... # Remplir ici !
3
4  def main():
5      print('Calcul du coût d\'un voyage en voiture !')
6      print('-------------------------------------')
7      print()
8      w = float(input('Consommation au 100 km : '))
9      x = float(input('Distance à parcourir : '))
10     y = float(input('Prix du carburant : '))
11     z = float(input('Coût du péage : '))
12
13     cout = calc_cout(w, x, y, z)
14
15     print()
16     print('Coût total du voyage :', cout)
17
18 main()
```

Exercice

Test :

```
>>> %Run cout_voyage_4.py
Calcul de coût d'un voyage en voiture !
---------------------------------------
Consommation au 100 km : 4.57
Distance à parcourir : 613
Prix du carburant : 1.62
Coût du péage : 32.4

Coût total du voyage : 155.56568400000003
```

Cours 3

Conditions

Précédemment sur ALGO1

Nous avons vu :

- Instruction d'affectation :

```
>>> x = 1
>>> x = x + 1
>>> x
2
```

- Définition et appel des fonctions :

succ.py

```
1 def succ(x=0):
2     return x + 1
```

```
>>> %Run succ.py
>>> succ(x=2)
3
```

Précédemment sur ALGO1

- Instruction input :

```
>>> ch = input('Entrez une donnée : ')
Entrez une donnée : test
>>> ch
'test'
```

- Conversions de valeurs :

```
>>> ch1 = input('Entrez un entier : ')
Entrez un entier : 10
>>> x = int(ch1)
>>> x + 1
11
>>> ch2 = input('Entrez un flottant : ')
Entrez un flottant : 5.5
>>> y = float(ch2)
>>> y + 3.3
8.8
```

Précédemment sur ALGO1

- Instruction print :

```
>>> peage = 32.4
>>> print(peage)
32.4
>>> print('Coût du péage :')
Coût du péage
```

- Chaînes de caractères
- Calcul du coût d'un voyage en voiture :

```
>>> %Run cout_voyage_4.py
Calcul de coût d'un voyage en voiture !
---------------------------------------
Consommation au 100 km : 4.57
Distance à parcourir : 613
Prix du carburant : 1.62
Coût du péage : 32.4

Coût total du voyage : 155.56568400000003
```

Aujourd'hui

Nous allons construire un programme qui demande à l'utilisateur d'entrer une année et ensuite signale si l'année est bissextile.

Exemple d'exécution :

```
Entrer une année: 2019
L'année 2019 n'est pas bissextile.
```

Sommaire

Section 13

Commentaires

Commentaires

Les **commentaires** servent à donner des explications aux personnes qui lisent le programme.

Exemple :

```
1 # Ceci est un commentaire, car commence avec # et
2 # Python va l'ignorer.
3
4 print('Ceci est une instruction,')
5 print('Python va l'afficher.')
```

```
Ceci est une instruction,
Python va l'afficher.
```

Commentaires

Il est possible d'insérer un commentaire à la fin d'une ligne d'instruction :

```
1 print('Salut !')    # Commentaire en fin de ligne.
```

```
Salut !
```

Commentaires

L'insertion des commentaires dans les programmes qui sont sauvegardés est une **bonne pratique de programmation**.

Cela permet de documenter et rendre plus compréhensibles ces programmes pour les humains.

Extrait d'un programme commenté :

cout_voyage_4.py

```
1  # Programme qui calcule le coût d'un voyage.
2  #
3  # Auteur: Tiago de Lima <tiago.delima@univ-artois.fr>
4  # Date: 23 jul 2014
5
6  def calc_cout(conso=4.57, dist=613, prix_carb=1.62,
       peage=32.4):
7      cout_carb = conso * (dist / 100) * prix_carb
8      total = (cout_carb + peage) * 2
9      return total
10
11 def main():
12     ...
13
```

Section 14

Expressions booléennes

Valeurs booléennes

Il existe deux **valeurs booléennes** (bool) en Python :

- vrai : `True`
- faux : `False`

Par exemple, les comparaisons renvoient des valeurs booléennes :

```
>>> 0 < 1
True
>>> 1 < 0
False
>>> 'une chaîne' < 'une autre chaîne'
False
```

Note : Python compare les chaînes de caractères selon l'ordre alphabétique.

Opérateurs de comparaison

Il y a plusieurs **opérateurs de comparaison** en Python :

opération	**opérateur**	**exemple**
inférieur	<	`0 < 1`
supérieur	>	`1 > 0`
égalité	==	`1 == 1`
supérieur ou égal	>=	`1 >= 1`
inférieur ou égal	<=	`0 <= 0`
différent	!=	`0 != 1`

Attention : **ne pas confondre l'instruction d'affectation avec l'opérateur d'égalité !**

```
>>> 1 = 1
File "<stdin>", line 1
    SyntaxError: keyword can't be an expression
>>> 1 == 1
True
```

Opérateurs booléens

Il y a trois **opérateurs booléens** en Python :

opération	**opérateur**	**exemple**
conjonction	`and`	`0 < 1 and 1 < 2`
disjonction	`or`	`0 < 1 or 1 > 0`
négation	`not`	`not 1 < 0`

Exemples :

```
>>> a = 0
>>> b = 1
>>> a < b and b < 2
True
>>> a < b or b < a
True
>>> not 1 < 0
True
```

Tables de vérité

not	
opération	résultat
`not False`	`True`
`not True`	`False`

and	
opération	résultat
`False and False`	`False`
`False and True`	`False`
`True and False`	`False`
`True and True`	`True`

or	
opération	résultat
`False or False`	`False`
`False or True`	`True`
`True or False`	`True`
`True or True`	`True`

Section 15

Conditionnels

Instruction `if`

L'**instruction `if`** permet de tester une certaine condition et de modifier le comportement du programme en conséquence.

Syntaxe :

```
if expr_1:
    suite_1
elif expr_2:
    suite_2
⋮
else:
    suite_n
```

Note : La partie `elif` et la partie `else` sont optionnelles.

L'instruction `if`

Exemple :

test_1.py

```
1 def test_1():
2     if 0 < 1:
3         ch = 'premier'     # ceci sera exécuté
4     elif 0 > 1:
5         ch = 'second'      # ne sera pas exécuté
6     else:
7         ch = 'troisième'   # ne sera pas exécuté
8     return ch
```

```
>>> %Run test_1.py
>>> test_1()
'premier'
```

L'instruction `if`

Exemple :

test_2.py

```
1 def test_2():
2     if 1 < 0:
3         ch = 'premier'     # ne sera pas exécuté
4     elif 1 > 0:
5         ch = 'second'      # ceci sera exécuté
6     else:
7         ch = 'troisième'   # ne sera pas exécuté
8     return ch
```

```
>>> %Run test_2.py
>>> test_2()
'second'
```

L'instruction `if`

Exemple :

test_3.py

```
1 def test_3():
2     if 0 < 0:
3         ch = 'premier'       # ne sera pas exécuté
4     elif 0 > 0:
5         ch = 'second'        # ne sera pas exécuté
6     else:
7         ch = 'troisième'     # ceci sera exécuté
8     return ch
```

```
>>> %Run test_3.py
>>> test_3()
'troisième'
```

Indentation

Attention : **la mise en page est très importante !**

Chaque instruction indentée sous l'instruction `if` est exécuté seulement si le teste est vrai :

```
1 if a < 1:
2     print('Si a est inférieur a 1, affiche ceci')
3     print('ainsi que ceci')
4 print('Ceci sera toujours imprimé')
5 print('car pas indenté')
```

Le programme ci-dessous ne fonctionne pas :

```
1 if a < 1:
2     print('Indentation avec 4 espaces')
3   print('Indentation avec 2 espaces: erreur !')
4    print('Indentation avec 3 espaces: erreur !')
```

Elif et else

Que fait ce programme ?

msg_temp.py

```
1 def msg_temp(temp=0):
2     if temp > 25.0:
3         msg = 'Il fait chaud !'
4     else:
5         msg = 'Il ne fait pas chaud.'
6     return msg
```

Comment changer ce programme de sorte à ce qu'il affiche :

Il fait froid !

si la température est inférieure à 10 ?

Elif et else

Le voici :

msg_temp.py

```
1 def msg_temp(temp=0):
2     if temp > 25.0:
3         msg = 'Il fait chaud !'
4     elif temp < 10.0:
5         msg = 'Il fait froid !'
6     else:
7         msg = 'Il ne fait ni chaud ni froid.'
8     return msg
```

Comment changer ce programme de sorte à ce qu'il retourne :

Vous pourriez faire frire des oeufs sur le trottoir !

si la température est supérieure à 35 ?

Elif et else

Le voici :

msg_temp.py

```
1  def msg_temp(temp=0):
2      if temp > 35.0:
3          msg = 'Vous pouvez faire frire des oeufs sur' + \
4                'le trottoir !'
5      elif temp > 25.0:
6          msg = 'Il fait chaud !'
7      elif temp < 10.0:
8          msg = 'Il fait froid !'
9      else:
10         msg = 'Il ne fait ni chaud ni froid.'
11     return msg
```

Erreurs communs

Attention : **Ceci ne fonctionne pas :**

```
1 if a < b or < c:                          # Erreur !
2     print('a est inférieur a b ou c')
```

Attention : **Ceci ne fonctionne pas comme prévu :**

```
1 if a == 'Paul' or 'Marie': # Incorrect : toujours vrai !
2     print('a est Paul ou Marie')
```

Ceci fonctionne :

```
1 if a < b or a < c:                        # Correct.
2     print('a est inférieur a b ou c')
```

```
1 if a == 'Paul' or a == 'Marie':           # Correct.
2     print('a est égal a Paul ou Marie')
```

Utilisation des variables booléennes

```
1  a = 1 > 0     # a contient la valeur True
2  if a:
3      # Ceci sera exécuté
4      print('1 supérieur a 0')
5  else:
6      print('1 n\'est pas supérieur à 0')
7
8  b = 2 < 1     # b contient la valeur False
9  if a and b:
10     print('1 supérieur a 0 et')
11     print('b inférieur a 1')
12 else:
13     # Ceci sera exécuté
14     print('1 supérieur à 0 mais')
15     print('b n\'est pas supérieur à 1')
```

Exercice 1

Problème :

Écrire une fonction qui reçoit un nombre entier, représentant une année, en argument et retourne si l'année est bissextile.

Sont bissextiles les années :

- soit divisibles par 4 mais non divisibles par 100 ;
- soit divisibles par 400.

Exercice 1

Algorithme :

1. Si `annee` est divisible par 400 :
   1.1 Retourne « vrai ».
2. Sinon, si `annee` est divisible par 4 mais non divisible par 100 :
   2.1 Retourne « vrai ».
3. Sinon :
   3.1 Retourne « faux ».

Exercice 1

Programme :

bissextile.py

```
1  def est_bissextile(annee=0):
2      # si divisible par 400
3      if (annee % 400) == 0:
4          return True
5
6      # sinon, si divisible par 4 et non divisible par 100
7      elif (annee % 4) == 0 and (annee % 100) != 0:
8          return True
9
10     # sinon
11     else:
12         return False
```

Exercice 1

Programme : (version compacte)

bissextile.py

```
1 def est_bissextile(annee=0):
2     return ((anne % 400) == 0) or \
3             (((annee % 4) == 0) and (annee % 100) != 0)
```

Note : Le « backslash » est utilisé pour continuer l'instruction dans la ligne suivante.

Exercice 2

Problème :

Écrire un programme qui demande à l'utilisateur d'entrer une année. Ensuite, le programme affiche un message indiquant si l'année est bissextile ou non.

Exercice 2

Programme :

bissextile.py

```
1  def est_bissextile(annee=0):
2      return ((anne % 400) == 0) or \
3             (((annee % 4) == 0) and (annee % 100) != 0)
4
5  def main():
6      annee = int(input('Entrez une année: '))
7
8      if est_bissextile(annee):
9          print('L\'année', annee, 'est bissextile.')
10     else:
11         print('L\'année', annee, 'n\'est pas bissextile.')
12
13 main()
```

Cours 4

Itérations

Précédemment sur ALGO1

Nous avons vu :

- Instruction d'affectation
- Définition et appel des fonctions
- Instruction input
- Conversions de valeurs
- Instruction print
- Chaînes de caractères

Précédemment sur ALGO1

- Commentaires

cout_voyage_4.py

```
1  # Programme qui calcule le coût d'un voyage.
2  #
3  # Auteur: Tiago de Lima <tiago.delima@univ-artois.fr>
4  # Date: 23 jul 2014
5
6  def calc_cout(conso=4.57, dist=613, prix_carb=1.62,
       peage=32.4):
7      cout_carb = conso * (dist / 100) * prix_carb
8      total = (cout_carb + peage) * 2
9      return total
10
11 def main():
12     ...
13
14 main()
```

Précédemment sur ALGO1

- Expressions et opérateurs booléens

```
>>> a = 0
>>> b = 1
>>> a < b and b < 2
True
>>> a < b or b < a
True
>>> not 1 < 0
True
```

Précédemment sur ALGO1

- Instruction `if-elif-else`

test_1.py

```
1 def test_1():
2     if 0 < 1:
3         ch = 'premier'     # ceci sera exécuté
4     elif 0 > 1:
5         ch = 'second'      # ne sera pas exécuté
6     else:
7         ch = 'troisième'   # ne sera pas exécuté
8     return ch
```

Aujourd'hui. . .

Nous allons jouer à vingt et un.

Exemple d'exécution :

```
VINGT ET UN

L'objectif est de lancer de dés pour se rapprocher le plus
    possible de 21 sans dépasser ce total.

Nombre de dés à lancer (0 pour quitter) : 3
Vos lancées : 1  3  1
Somme : 5
Les lancées de la banque : 5  4  6
Somme : 15
Nombre de dés à lancer (0 pour quitter) : 3
Vos lancées : 5  4  2
Somme : 16
Les lancées de la banque : 5  6  3
Somme : 29
Vous avez gagné !
```

Sommaire

Section 16

Fonction range

Fonction range

La **fonction `range`** génère une séquence de nombres entiers.

Syntaxe :

```
range([exp_1], exp_2, [exp_3])
```

`exp_1` : début de l'intervalle

`exp_2` : fin de l'intervalle

`exp_3` : pas

Fonction range

Exemples :

```
>>> range(0, 3)
range(0, 3)
>>> type(range(0, 3))
<class 'range'>
>>> list(range(0, 3))
[0, 1, 2]
```

Notez que `range(0, 3)` n'est pas une liste. Toutefois, l'objet « range » peut être transformé en une liste.

L'objet généré par `range(0, 3)` contient les éléments $(0, 1, 2)$.

C'est-à-dire, `range(0, 3)` correspond aux entiers contenus dans l'intervalle $[0; 3[$.

Fonction range

Les autres formes de la fonction `range` :

```
>>> list(range(3))                  # [0; 3[
[0, 1, 2]
>>> list(range(3, 8))               # [3; 8[
[3, 4, 5, 6, 7]
>> list(range(3, 15, 2))            # [3; 15[ avec step 2
[3, 5, 7, 9, 11, 13]
>> list(range(20, 2, -5))           # ]2; 20] avec step -5
[20, 15, 10, 5]
>>> list(range(20, 0, -5))          # ]0; 20] avec step -5
[20, 15, 10, 5]
>>> list(range(0, 20, 5))           # [0; 20[ avec step 5
[0, 5, 10, 15]
```

Note : Notez que `list` est aussi une fonction de conversion de valeurs. Mais on vera plus de détails plus tard.

Section 17

Instruction for

Instruction for

L'**instruction `for`** sert à itérer une partie du programme un nombre prédéterminé de fois.

Syntaxe :

```
for var in liste:
    suite
```

Instruction for

Exemple :

```
1 def fonc():
2     for i in range(4):
3         print('Aidez-moi Obi-Wan Kenobi...')
```

```
>>> fonc()
Aidez-moi Obi-Wan Kenobi...
Aidez-moi Obi-Wan Kenobi...
Aidez-moi Obi-Wan Kenobi...
Aidez-moi Obi-Wan Kenobi...
```

La fonction `range` en ligne 1 sert à définir le nombre de fois que l'instruction se répétera. Ici, 4 fois.

La variable `i` en ligne 1 contient la valeur curante de chaque itération.

L'instruction `print` en ligne 2 est l'instruction qui se répéte 4 fois.

Instruction for

Encore une fois, **la mise en page est très importante !**

```
1 def fonc():
2     for i in range(4):
3         print('Encore une fois, ')
4     print('la mise en page est très importante !')
```

```
>>> fonc()
Encore une fois,
Encore une fois,
Encore une fois,
Encore une fois,
la mise en page est très importante !
```

Instruction for

Encore une fois. . . :

```
1 def fonc():
2     for i in range(4):
3         print('Encore une fois, ')
4         print('la mise en page est très importante !')
```

```
>>> fonc()
Encore une fois,
la mise en page est très importante !
Encore une fois,
la mise en page est très importante !
Encore une fois,
la mise en page est très importante !
Encore une fois,
la mise en page est très importante !
```

L'instruction `for`

Le rôle de la variable `i` :

```
1 def fonc():
2     for i in range(7):
3         print(i)
```

```
>>> fonc()
0
1
2
3
4
5
6
```

Il n'y a pas de contrainte supplémentaire sur le nom de la variable `i`.
Elle peut s'appeler `x`, `y`, `ligne`, etc..

Comptage

Afficher les nombres de 1 à 10 (version 1) :

```
1 def compte_dix_v1():
2     for x in range(1, 11):
3         print(x)
```

```
>>> compte_dix_v1()
1
2
3
4
5
6
7
8
9
10
```

Comptage

Afficher les nombres de 1 à 10 (version 2) :

```
1 def compte_dix_v2():
2     for x in range(10):
3         print(x + 1)
```

```
>>> compte_dix_v2()
1
2
3
4
5
6
7
8
9
10
```

Comptage

Deux manières d'afficher les nombres paires de 2 à 10 :

```
1 def paires_2_10_v1():
2     for z in range(2, 12, 2):
3         print(z)
```

```
1 def paires_2_10_v2():
2     for z in range(5):
3         print((z + 1) * 2)
```

Exécution :

```
>>> paires_2_10_v2()
2
4
6
8
10
```

Comptage

Compter de 5 à 1 (à l'envers) :

```
1 def compte_5_1():
2     for i in range(5, 0, -1):
3         print(i)
```

```
>>> compte_5_1()
5
4
3
2
1
```

Afficher le contenu d'une liste

```
1 def affiche_liste():
2     for i in [2, 6, 4, 2, 4, 6, 7, 4]:
3         print(i)
```

```
>>> affiche_liste()
2
6
4
2
4
6
7
4
```

Boucles imbriquées

Quelle est la difference entre ce deux programmes ?

```
1 def fonc_1():
2     for i in range(3):
3         print('a')
4     for j in range(3):
5         print('b')
```

```
1 def fonc_2():
2     for i in range(3):
3         print('a')
4         for j in range(3):
5             print('b')
```

Boucles imbriquées

Premier programme :

```
1 def fonc_1():
2     for i in range(3):
3         print('a')
4     for j in range(3):
5         print('b')
```

```
>>> fonc_1():
a
a
a
b
b
b
```

Boucles imbriquées

Deuxième programme :

```
1 def fonc_2():
2     for i in range(3):
3         print('a')
4         for j in range(3):
5             print('b')
```

```
>>> fonc_2()
a
b
b
b
a
b
b
b
a
b
b
b
```

Faire une somme

Calculer la somme de 1 à 100 :

```
1 def somme_1_100():
2     somme = 0
3     for i in range(1, 101):
4         somme = somme + i
5     return somme
```

```
>>> somme_1_100()
5050
```

Calculer le total

```
1 def somme():
2     total = 0                          # Initialise.
3     for i in range(5):                 # Repete 5 fois.
4         nombre = int(input('Entrez un nombre : '))
5         total = total + nombre          # Cumule.
6     return total                        # Retourne le total.
```

```
>>> somme()
Entrez un nombre : 1
Entrez un nombre : 2
Entrez un nombre : 3
Entrez un nombre : 4
Entrez un nombre : 5
15
```

Combien de zéros ?

Calculer le nombre de fois qu'un utilisateur a rentré l'entier 0 (zéro) :

```
1 def nombre_zeros():
2     total = 0                          # Initialise.
3     for i in range(5):                 # Repete 5 fois.
4         nombre = int(input('Entrez un nombre : '))
5         if nombre == 0:                # Si le nombre est 0:
6             total = total + 1          # Cumule.
7     return total
```

```
>>> nombre_zeros()
Entrez un nombre : 1
Entrez un nombre : 0
Entrez un nombre : 2
Entrez un nombre : 0
Entrez un nombre : 4
2
```

Exercice

Quelle valeur retourne cette fonction ?

```
1 def fonc():
2     a = 0
3     for i in range(3):
4         a = a + 1
5     return a
```

```
>>> fonc()
3
```

Exercice

Quelle valeur retourne cette fonction ?

```
1 def fonc():
2     a = 0
3     for i in range(3):
4         a = a + 1
5     for j in range(3):
6         a = a + 1
7     return a
```

```
>>> fonc()
6
```

```
1 def fonc():
2     a = 0
3     for i in range(3):
4         a = a + 1
5         for j in range(3):
6             a = a + 1
7     return a
```

```
>>> fonc()
12
```

Section 18

Instruction while

Instruction while

L'**instruction `while`** sert à itérer une partie du programme tant qu'une certaine condition soit satisfaite.

En principe, nous ne savons pas en avance combien d'itérations seront necéssaires.

Syntaxe :

```
while expr :
    suite
```

Instruction while

Exemple :

```
1 def fonc():
2     x = int(input('Entrez un nombre : '))
3     i = 0
4     while i < x:
5         print(i)
6         i = i + 1
```

```
>>> fonc()
Entrez un nombre : 5
0
1
2
3
4
```

Instruction while

Attention : `while` ne s'utilise pas avec `range`.

Ce programme ne fonctionne pas :

```
1 def fonc():
2     while range(5):
3         print(i)
```

```
Traceback (most recent call last):
    File "<stdin>", line 2, in <module>
    NameError: name 'i' is not defined
```

Écriture compacte des opérations

Par exemple, l'**opérateur d'incrément** est souvent utilisé dans les boucles `while` :

```
1 def fonc():
2     x = int(input('Entrez un nombre : '))
3     i = 0
4     while i < x:
5         print(i)
6         i += 1
```

L'instruction en ligne 6 est équivalente à : `i = i + 1`.

La même chose peut être faite avec les autres opérateurs arithmétiques :

opérateur	exemple	opération
`+=`	`i += 1`	`i = i + 1`
`-=`	`i -= 1`	`i = i - 1`
`*=`	`i *= 2`	`i = i * 2`
`/=`	`i /= 2`	`i = i / 2`

Boucler jusqu'à ce que l'utilisateur veuille quitter

```
1 def fonc():
2     quitter = 'non'
3     while quitter == 'non':
4         quitter = input('Voulez-vous quitter ? ')
```

```
>>> fonc()
Voulez-vous quitter ? non
Voulez-vous quitter ? non
Voulez-vous quitter ? non
Voulez-vous quitter ? oui
```

Boucler jusqu'à ce que le jeu soit terminé

Version 1 (erronée) :

```
1  def dragon():
2      terminer = False
3      while not terminer:
4          quitter = input('Voulez-vous quitter ? ')
5          if quitter == 'oui':
6              terminer = True
7
8          att = input('Voulez-vous attaquer le dragon ? ')
9          if att == 'oui':
10             print('Mauvais choix, vous êtes mort !')
11             terminer = True
```

Cette version n'est pas correcte, parce que quand le joueur décide de quitter, le programme demande quand-même s'il veut attaquer le dragon.

Comment faire pour le corriger ?

Boucler jusqu'à ce que le jeu soit terminé

Version 2 (corrigée) :

```
1  def dragon():
2      terminer = False
3      while not terminer:
4          quitter = input('Voulez-vous quitter ? ')
5          if quitter == 'oui':
6              terminer = True
7          else:
8              att = input('Voulez-vous attaquer le dragon ?
                   ')
9              if att == 'oui':
10                 print('Mauvais choix, vous êtes mort !')
11                 terminer = True
```

Erreurs communs

Nous voulons écrire un programme qui compte à l'envers à partir de 10. Pourtant, ceci ne fonctionne pas. Pourquoi ?

```
1 def fonc():
2     i = 10
3     while i == 0:
4         print(i)
5         i -= 1
```

Maintenant, nous voulons écrire un programme qui compte jusqu'à 10. Pourtant, ceci ne fonctionne pas. Pourquoi ?

```
1 def fonc():
2     i = 1
3     while i < 10:
4         print(i)
```

Exercice

Problème : Programmez le jeu Vingt et Un.

Il s'agit d'un jeu qui oppose un joueur à la banque. L'objectif est de lancer de dés pour se rapprocher le plus possible de 21 sans dépasser ce total. À chaque tour du jeu, le joueur peut lancer le nombre de dés qu'il souhaite. Le total est calculé. Ensuite, la banque lance le même nombre de dés. Celui qui dépasse 21 a perdu la partie. Si les deux dépassent 21, la partie est nulle. Sinon, le joueur peut continuer à lancer des dés pour continuer à se rapprocher 21.

Exercice

Exemple d'exécution :

```
VINGT ET UN

L'objectif est de lancer de dés pour se rapprocher le plus
    possible de 21 sans dépasser ce total.

Nombre de dés à lancer (0 pour quitter) : 3
Vos lancées : 1  3  1
Somme : 5
Les lancées de la banque : 5  4  6
Somme : 15
Nombre de dés à lancer (0 pour quitter) : 3
Vos lancées : 5  4  2
Somme : 16
Les lancées de la banque : 5  6  3
Somme : 29
Vous avez gagné !
```

Exercice

Cahier de charge : Nous allons réaliser ce jeu en plusieurs fonctions :

- `affiche_ouverture` : affiche l'ouverture du jeu.
- `calcule_lancee` : reçoit un entier `nombre_des` en argument, affiche les résultats des dés d'une lancée et retourne la valeur total de la lancée.
- `calcule_gagnant` : reçoit deux entiers `joueur` et `banque` en argument, retourne l'eventuel gagnant du match.
- `main` : ne reçoit pas d'arguments. C'est la fonction principale du programme.

Exercice

```
1 def affiche_ouverture():
2     print()
3     print('VINGT ET UN')
4     print('L'objectif ... ')
5     print()
```

Exercice

```
1  from random import randint
2
3  ...
4
5  def calcule_lancee(nombre_des=1):
6      total = 0
7      for i in range(nombre_des):
8          valeur = randint(1, 6)
9          total += valeur
10         print(valeur, end=' ')
11     return total
```

Exercice

```
1  VALEUR = 21
2  JOUEUR_GAGNE = 0
3  JOUEUR_PERDU = 1
4  MATCH_NUL = 2
5  PAS_FINI = 3
6
7  ...
8
9  def calcule_gagnant(joueur=0, banque=0):
10     if joueur > VALEUR:
11         if banque <= VALEUR:
12             return JOUEUR_PERDU
13         else:
14             return MATCH_NUL
15     else:
16         if banque > VALEUR:
17             return JOUEUR_GAGNE
18         else:
19             return PAS_FINI
```

Exercice

```
1  def main():
2      affiche_ouverture()
3
4      somme_joueur = 0
5      somme_banque = 0
6      termine = False
7      while not termine:
8
9          # Demande le nombre de dés à lancer.
10         nombre_des = int(input('Nombre de dés à lancer (0
               pour quitter) : '))
11
12         # Si nombre de dés à lancer est 0, fini la partie.
13         if nombre_des == 0:
14             termine = True
15
16         else:
17             # Calcule les lancées du joueur.
18             print('Vos lancées :', end=' ')
19             somme_joueur += calcule_lancee(nombre_des)
20             print('\nSomme :', somme_joueur)
21             print()
```

Exercice

```
22              # Calcule les lancées de la banque.
23              print('Les lancées de la banque :', end=' ')
24              somme_banque += calcule_lancee(nombre_des)
25              print('\nSomme :', somme_banque)
26              print()
27
28              # Calcule le gangnant.
29              g = calcule_gagnant(somme_joueur,
                    somme_banque)
30              if g == JOUEUR_GAGNE:
31                  print('Vous avez gagné !')
32                  termine = True
33
34              elif g == JOUEUR_PERDU:
35                  print('Vous avez perdu.')
36                  termine = True
37
38              elif g == MATCH_NUL:
39                  print('Match nul.')
40                  termine = True
```

Exercice

Programme : Voir fichier sur Moodle : `vingt_et_un.py`

Cours 5

Fonctions et modules

Précédemment sur ALGO1

Nous avons vu :

- Instruction d'affectation
- Définition et appel des fonctions
- Instruction `input`
- Conversions de valeurs
- Instruction `print`
- Chaînes de caractères
- Commentaires
- Expressions et opérateurs booléens
- Instruction `if`-`elif`-`else`

Précédemment sur ALGO1

- La fonction `range` :

```
>>> list(range(3))
[0, 1, 2]
>>> list(range(3, 8))
[3, 4, 5, 6, 7]
>> list(range(3, 15, 2))
[3, 5, 7, 9, 11, 13]
>> list(range(20, 2, -5))
[20, 15, 10, 5]
>>> list(range(20, 0, -5))
[20, 15, 10, 5]
>>> list(range(0, 20, 5))
[0, 5, 10, 15]
```

Précédemment sur ALGO1

- L'instruction `for` :

```
1 def fonc():
2     for i in range(4):
3         print('Aidez-moi Obi-Wan Kenobi...')
```

```
>>> fonc()
Aidez-moi Obi-Wan Kenobi...
Aidez-moi Obi-Wan Kenobi...
Aidez-moi Obi-Wan Kenobi...
Aidez-moi Obi-Wan Kenobi...
```

Précédemment sur ALGO1

- L'instruction `while` :

```
1 def fonc():
2     x = int(input('Entrez un nombre : '))
3     i = 0
4     while i < x:
5         print(i)
6         i = i + 1
```

```
>>> fonc()
Entrez un nombre : 5
0
1
2
3
4
```

Aujourd'hui . . .

Nous allons créer un menu d'options.

```
MENU :
    a. Calculer le volume d'une sphère.
    b. Calculer le volume d'un cylindre.
    q. Quitter.

Entrez votre option : a

Entrez le rayon de la sphère : 34
Volume : 164636.21020892428

MENU :
    a. Calculer la volume d'une sphère.
    b. Calculer le volume d'un cylindre.
    q. Quitter.

Entrez votre option : q
```

Sommaire

Section 19

Définition et appel de fonctions

Définition des fonctions

Les **fonctions** servent à deux choses :

- rendre le programme plus lisible et organisé
- permettre la réutilisation du code

Syntaxe de la définition d'une fonction :

```
def nom(arg_1, arg_2, ..., arg_n):
    suite
```

`nom`: le nom de la fonction

`arg_i`: une variable qui stockera les valeurs passées en argument

`suite`: une suite d'instructions

Syntaxe de l'appel d'une fonction :

```
nom(arg_1, arg_2, ..., arg_n)
```

Définition des fonctions

Exemple :

```
1  def volume_sphere(rayon):
2      pi = 3.1415
3      volume = 4 / 3 * pi * rayon ** 3
4      return volume
5
6  def main():
7      print('Le volume de trois sphères :')
8      print('r =  3, vol =', volume_sphere(3))
9      print('r = 10, vol =', volume_sphere(10))
10     print('r = 50, vol =', volume_sphere(50))
```

```
>>> main()
Le volume de trois sphères :
r =  3, vol = 113.09400000000001
r = 10, vol = 4188.790204786391
r = 50, vol = 523598.7755982988
```

Fonctions à plusieurs arguments

Exemple :

```
1  def volume_cylindre(rayon, hauteur) :
2      pi = 3.1415
3      volume  = pi * rayon ** 2 * hauteur
4      return volume
5
6  def main():
7      print('Le volume de trois cylindres :')
8      print('r = 12, h = 3, vol =', volume_cylindre(12, 3))
9      print('r = 22, h = 8, vol =', volume_cylindre(22, 8))
10     print('r = 11, h = 7, vol =', volume_cylindre(11, 7))
```

```
>>> main()
Le volume de trois cylindres :
r = 12, h = 3, vol = 1357.1680263507906
r = 22, h = 8, vol = 12164.24675469968
r = 11, h = 7, vol = 2660.928977590555
```

Afficher vs. retourner la valeur

```
1  def sum_print(a, b):
2      result = a + b
3      print(result)
4
5  def sum_return(a, b):
6      result = a + b
7      return result
8
9  def main():
10     sum_print(4, 4)         # Affiche "8".
11     sum_return(4, 4)        # Pas d'affichage.
12     x1 = sum_print(4, 4)    # Affiche "8", affecte None
13     x2 = sum_return(4, 4)   # Affecte 8 à x2
14     print('x1 =', x1)       # Affiche "x1 = None"
15     print('x2 =', x2)       # Affiche "x2 = 8"
```

```
>>> main()
8
8
x1 = None
x2 = 8
```

Afficher vs. retourner la valeur

La fonction doit retourner la valeur quand nous voulons la sauvegarder.

La fonction doit afficher la valeur quand nous voulons la voir à l'écran.

Une fonction peut aussi afficher et retourner la valeur.

Dans ce dernier cas, il faut faire l'affichage d'abord.

Documentation des fonctions

Il est recommandé de documenter les fonctions.

En Python, cela se fait avec une chaîne de caractères immédiatement après la définition de la fonction :

Exemple :

```
1  def volume_cylindre(rayon, hauteur) :
2      ''' Calcule le volume d'un cylindre.
3      Arguments:
4          rayon : float -- rayon de la base du cylindre.
5          hauteur : float -- hauteur du cylindre.
6      Retour:
7          float -- volume du cylindre. '''
8      pi = 3.1415
9      volume  = pi * rayon ** 2 * hauteur
10     return volume
```

ATTENTION : à partir de ce moment, ceci devient obligatoire en TP, dans le PROJET, en DS et en EXAMEN !

Section 20

Porté de variables et de paramètres

Porté de variables

La **porté** d'une variable est la partie du programme où la variable est disponible.

Exemple 1 :

```
1 def f():
2     x = 22              # Définition de x.
3     print('x =', x)     # Affiche la valeur de x.
4
5 def main():
6     f()
7     print('x =', x)     # Erreur : x pas disponible ici.
```

```
>>> main()
x = 22
Traceback (most recent call last):
    File "<pyshell>", line 1, in <module>
    NameError: name 'x' is not defined
```

Porté de variables

Exemple 2 :

```
1 x = 44                    # Définition globale de x.
2
3 def f():
4     print('x =', x)       # x est disponible ici.
5
6 def main():
7     f()
8     print('x =', x)       # x est disponible ici.
```

```
>>> main()
x = 44
x = 44
```

Porté de variables

Les variables **locales** sont prioritaires sur les variables **globales**.

Exemple 3 :

```
1 x = 44                    # Définition globale de x.
2
3 def f():
4     x = 55                # Définition locale de x.
5     print('x =', x)       # Affiche : x = 55
6
7 def main():
8     f()
9     print('x =', x)       # Affiche : x = 44
```

```
>>> main()
x = 55
x = 44
```

Porté de variables

Confusion entre variable locale et variable globale.

Exemple 4 :

```
1 x = 44                    # Définition globale de x.
2
3 def f() :
4     x = x + 1             # Erreur : quelle x utiliser ?
5     print('x =', x)
```

```
>>> f()
Traceback (most recent call last):
    File "<pyshell>", line 1, in <module>
    ...
    UnboundLocalError: local variable 'x' referenced before
    assignment
```

Les variables globales

Python permet l'insertion des instructions au niveau d'indentation 0. C'est ce qu'on a fait pour l'instant.

Cela veut dire, par exemple, que nos variables sont **globales**. Elles sont donc accessibles partout.

Les variables globales sont souvent une **mauvaise idée** parce que n'importe quelle partie du programme peut les changer. (Imaginez un programme avec 50 000 lignes de code ! !)

Il faut donc **limiter l'usage des variables globales.**

Section 21

Arguments

Arguments

Le passage d'arguments en Python est fait par copie.

Cela veut dire que les arguments contiennent une copie de la valeur passée en argument.

Exemple :

```
1 def f(x):
2     x = x + 1          # Utilisation de l'argument.
3     print('x =', x)    # Affiche 'x = 11'.
4
5 def main():
6     y = 10
7     f(y)               # Passage de y en argument.
8     print('y =', y)    # Affiche 'y = 10'.
```

```
>>> main()
x = 11
y = 10
```

Arguments

```
1 def f(x):
2     x = x + 1          # Utilisation de l'argument.
3     print('x =', x)    # Affiche 'x = 11'.
4
5 def main():
6     x = 10             # Définition d'un autre x.
7     f(x)               # Passage de x en argument.
8     print('x =', x)    # Affiche 'x = 10'.
```

```
>>> main()
x = 11
x = 10
```

En effet, il y a deux variables différentes qui s'appellent `x` :

- une dans la fonction
- une à l'extérieur de la fonction

Appels à des fonctions dans des fonctions

Il est tout à fait possible d'appeler une fonction dans une fonction.

Exemple :

```
1 def surface_carre(largeur) :
2     return largeur * largeur
3
4 def volume_cube(largeur) :
5     return surface_carre(largeur) * largeur
6
7 def main():
8     print(volume_cube(3))
9     print(volume_cube(5))
```

```
>>> main()
27
125
```

Section 22

Exercice

Exercice

Problème :

Utiliser les fonctions pour écrire un programme qui demande à l'utilisateur quel calcul il souhaite réaliser :

- le volume d'une sphère
- le volume d'un cylindre

Selon l'option choisi, le programme demande à l'utilisateur les données du calcul et imprime le résultat.

Exercice

Cahier de charge :

Après une analyse du problème, nous décidons d'écrire le programme à l'aide de plusieurs fonctions :

- `volume_sphere(float) -> float`
  Retourne le volume de la sphère dont le rayon est passé en argument.
- `volume_cylindre(float, float) -> float`
  Retourne le volume d'un cylindre dont le rayon et la hauteur sont passés en argument.
- `print_menu(None) -> None`
  Affiche le menu d'options.
- `main(None) -> None`
  La fonction principale du programme.

La fonction `print_menu`

```
1 def print_menu():
2     ''' Affiche le menu d'options. '''
3     print('''
4 MENU :
5     a. Calcule le volume d\'une sphère.
6     b. Calcule le volume d\'un cylindre.
7     q. Quitte.
8 ''')
```

La fonction `main`

```
def main():
    ''' Fonction principale du programme.
        ...'''
    quitter = False
    while not quitter:
        print_menu()
        option = input('Entrez votre option : ')
        if option == 'a':
            r = float(input('Rayon de la sphere : '))
            print('Volume :', volume_sphere(r))
        elif option == 'b':
            r = float(input('Rayon de la base : '))
            h = float(input('Hauteur du cylindre : '))
            print('Volume :', volume_cylindre(r, h))
        elif option == 'q':
            quitter = True
        else:
            print('Option invalide.')
```

Exercice

Programme complet dans le fichier `menu.py`.

Test :

```
MENU :
    a. Calculer le volume d'une sphère.
    b. Calculer le volume d'un cylindre.
    q. Quitte.

Entrez votre option : b
Rayon de la base : 2
Hauteur du cylindre : 3
Volume : 62.83185307178

MENU :
    a. Calculer le volume d'une sphère.
    b. Calculer le volume d'un cylindre.
    q. Quitte.

Entrez votre option : q
```

Section 23

Modules

Modules

Lors de la confection d'un nouveau programme, il nous arrive souvent de devoir réutiliser des fonctions que nous avons déjà créé.

Une option est de faire un copier-coller des anciennes fonctions dans le nouveau programme.

Une option plus pratique est d'utiliser les **modules**.

Un **module** en python est un fichier qui contient des fonctions pré-définies et qui peut être **importé** par d'autres programmes et autres modules.

Le module chaines.py

chaines.py

```
1  ''' Chaînes
2
3  Collection des fonctions de traitement de
4  chaînes de caractères. '''
5
6  def appartient(car, ch):
7      ...
8
9  def est_numerique(ch):
10     ...
11
12 def minuscule(ch):
13     ...
```

Importation d'un module dans un programme

```
1  from chaines import appartient
2
3  def main():
4      chaine = input('Entrez une chaîne : ')
5      lettre = input('Entrez une lettre : ')
6      if appartient(lettre, chaine):
7          print('La lettre appartient à la chaîne.')
8      else:
9          print('La lettre n\'appartient pas à la chaîne.')
10
11 # Appel à la fonction principale.
12 main()
```

```
Entrez une chaîne : abcde
Entrez une lettre : a
La lettre appartient à la chaîne.
```

Importation d'un module dans un autre module

inputs.py

```
''' Inputs

Collection des fonctions d'entrée qui
testent si l'utilisateur à donné une entrée valide,
sinon redemande. '''

from chaines import appartient

def input_options(message, options):
    ...

def input_int(message):
    ...

def input_float(message):
    ...
```

Utilisation d'un module

Pour utiliser le module, il suffit de l'importer dans le programme :

```
1 from inputs import input_int
2 ...
```

Dans cet exemple, seulement la fonction `input_int` sera visible.

Les autres fonctions, y compris les fonctions du module `chaines` ne sont pas visibles dans le programme.

Il est possible d'importer plusieurs modules et plusieurs fonctions de chaque module :

```
1 from math import *
2 from inputs import input_int, input_float
```

Section 24

La fonction principale du programme (`main`)

La fonction `main`

Nous allons donc créer une **fonction principale** pour notre programme.

Pour des questions d'habitude, nous allons l'appeler `main`.

Nous allons aussi conditionner l'appel à la fonction `main`.

menu_options.py

```
1 def main() :
2     # ... instructions du programme
3
4 if __name__ == '__main__' :
5     main()
```

Les deux dernières lignes servent à appeler la fonction `main`, seulement dans le cas où ce fichier est exécuté, et pas « importé » par un autre programme.

Les fichiers du programme

Voici à quoi ressemble le dossier qui contient le programme.

```
algo1/
 |
 + menu_options/
    |
    + menu_options.py
    |
    + inputs.py
    |
    + chaines.py
```

Pour exécuter le programme, vous devez exécuter (« run ») le fichier qui contient la fonction principale `main`.

Sous linux, vous pouvez également utiliser la ligne de commande :

```
python3 algo1/menu_options/menu_options_2.py
```

Les fichiers du programme

Une option encore plus pratique est de renommer le fichier principal pour que Python sache comment exécuter le programme à partir du dossier.

```
algo1/
 |
 + menu_options/
    |
    + __main__.py
    |
    + inputs.py
    |
    + chaines.py
```

Pour exécuter le programme, vous devez exécuter (« run ») le fichier `__main__.py`, qui contient la fonction principale `main`.

Mais sous linux, vous pouvez également utiliser la commande :

```
python3 algo1/menu_options/
```

Cours 6

Types composites

Précédemment sur ALGO1

Nous avons vu :

- Instruction d'affectation
- Définition et appel des fonctions
- Instruction `input`
- Conversions de valeurs (fonctions `int` et `float`)
- Instruction `print`
- Chaînes de caractères
- Commentaires
- Expressions et opérateurs booléens
- Instruction `if`-`elif`-`else`
- Fonction `range`
- Instruction `for`
- Instruction `while`

Précédemment sur ALGO1

Documentation de fonctions.

```
1  def volume_cylindre(rayon, hauteur) :
2      ''' Calcule le volume d'un cylindre.
3      Arguments:
4          rayon : float -- rayon de la base du cylindre.
5          hauteur : float -- hauteur du cylindre.
6      Retour:
7          float -- volume du cylindre. '''
8      pi = 3.1415
9      volume  = pi * rayon ** 2 * hauteur
10     return volume
```

ATTENTION : à partir de ce moment, ceci devient obligatoire en TP, dans le PROJET, en DS et en EXAMEN !

Précédemment sur ALGO1

Porté de variables :

```
1 x = 44                    # Définition globale de x.
2
3 def f():
4     x = 55                # Définition locale de x.
5     print('x =', x)       # Affiche : x = 55
6
7 def main():
8     f()
9     print('x =', x)       # Affiche : x = 44
```

```
>>> main()
x = 55
x = 44
```

Précédemment sur ALGO1

Les modules et programmes en plusieurs fichiers :

```
algo1/
 |
 + menu_options/
    |
    + __main__.py
    |
    + inputs.py
    |
    + chaines.py
```

Sommaire

Section 25

Séquences

Listes

Une **liste** est une suite des valeurs.

Syntaxe :

```
[ <expr1>, <expr2>, ..., <exprN> ]
```

En Python, les listes peuvent être hétérogènes (c.-à-d., peuvent contenir des valeurs de plusieurs types différents.)

Exemples :

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

```
>>> jours = ['lundi', 'mardi', 3, 4, 5.59]
>>> jours
['lundi', 'mardi', 3, 4, 5.59]
```

Listes

Nous pouvons accéder aux éléments d'une liste individuellement.

```
>>> jours[0]
'lundi'
>>> jours[3]
4
```

Attention :

- **Le premier élément d'une liste est l'élément 0 !**
- **Le dernier élément d'une liste de n éléments est l'élément $n-1$!**

Nous pouvons aussi remplacer un élément dans une liste :

```
>>> l = [0, 1, 2, 3]
>>> l[0] = 55
>>> l
[55, 1, 2, 3]
>>> l[3] = 4.44
>>> l
[55, 1, 2, 4.44]
```

Opérateurs de liste

Les opérateurs de chaîne de caractères fonctionnent aussi avec les séquences :

```
>>> l = [0, 1, 2, 3]
>>> l + l
[0, 1, 2, 3, 0, 1, 2, 3]
```

```
>>> l[-1]
3
```

```
>>> 3 * l
[0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3]
```

Fonctions et méthodes de listes

La fonction `len` retourne la longueur d'une liste :

```
>>> l = [0, 1, 2, 3]
>>> len(l)
4
```

La fonction `del` supprime un élément de la liste :

```
>>> del(l[2])
>>> l
[0, 1, 3]
>>> l[2]
3
```

La méthode `append` ajoute un élément à la fin de la liste :

```
>>> l.append(99)
>>> l
[0, 2, 3, 99]
```

Ajouts dans une liste

Pour ajouter un élément à la **fin** de la liste, nous pouvons faire :

```
>>> l = [0, 1, 2, 3]
>>> l = l + [99]
>>> l
[0, 1, 2, 3, 99]
```

Pour ajouter un élément au **début** d'une liste, nous pouvons faire :

```
>>> l = [0, 1, 2, 3]
>>> l = [99] + l
>>> l
[99, 0, 1, 2, 3]
```

Note : en effet, ces opérations créent une nouvelle liste `l` avec un élément de plus.

Ajouts dans une liste

Pour ajouter un élément à la liste sans la création d'une nouvelle liste, nous devons utiliser la méthode `append`, comme vu auparavant.

```
>>> l = [0, 1, 2, 3]
>>> l.append(99)
>>> l
[0, 1, 2, 3, 99]
```

Tuples

Un **tuple** est une liste immuable.

Syntaxe :

```
(<expr1>, <expr2>, ..., <exprN>)
```

Exemples :

```
>>> (1, 2, 3)
(1, 2, 3)
>>> mois = ('janvier', 'fevrier', 3, 4, 6.77)
>>> mois
('janvier', 'fevrier', 3, 4, 6.77)
```

Tuples

La différence entre les listes et le tuples est que les dernières sont immuables :

```
>>> l = [1, 2]
>>> t = (1, 2)
>>> l[0] = 33
>>> l
[33, 2]
>>> t[0] = 22
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    TypeError: 'tuple' object does not support item assignment
```

Donc, évidement, la fonction `del` ne fonctionnent pas non plus avec les tuples.

L'avantage des tuples par rapport aux listes est la performance. Python est plus performant avec les tuples.

Section 26

Exemples avec séquences

Itération sur séquences (version 1)

```
1 def print_seq_v1(s):
2     ''' ... '''
3     for i in range(len(s)) :  # pour i de 0 à len(s) - 1:
4         print(s[i])           #   affiche l'élém. i de s
```

```
>>> l = [1, 2, 3]
>>> t = (1, 2, 3)
>>> print_seq_v1(l)
1
2
3
>>> print_seq_v1(t)
1
2
3
```

Itération sur séquences (version 2)

L'instruction `for` peut être utilisée directement avec les séquences :

```
1 def print_seq_v2(s):
2     ''' ... '''
3     for e in s :     # pour chaque élément e de s:
4         print(e)     #   affiche e
```

```
>>> l = [1, 2, 3]
>>> t = (1, 2, 3)
>>> print_seq_v2(l)
1
2
3
>>> print_seq_v2(t)
1
2
3
```

Création de listes

Création d'une liste de nombres de 0 à $n - 1$:

```
def cree_liste(n):
    ''' ... '''
    l = []                      # l = liste vide
    for i in range(n) :         # pour i de 0 à n - 1:
        l.append(i + 1)         #   ajoute i dans l
    return l
```

```
>>> ma_liste = cree_liste(50)
>>> ma_liste
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
    17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30,
    31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44,
    45, 46, 47, 48, 49]
```

Création de listes

Création d'une liste avec *n* fois le nombre *x* :

```python
def cree_liste(n, x):
    ''' ... '''
    l = []                 # l = liste vide
    for i in range(n):     # pour i de 0 à n - 1:
        l.append(x)        #   ajoute x dans l
    return l
```

```
>>> ma_liste = cree_liste(100, 0)
>>> ma_liste
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Création de listes

Création d'une liste contenant *n* nombres entrés par l'utilisateur :

```
1 def input_liste(n):
2     ''' ... '''
3     l = []                      # l = liste vide
4     for i in range(n) :         # pour i de 0 à n - 1:
5         x = float(input('Entrez un nombre : '))
6         l.append(x)             #   ajoute x dans l
7     return l
```

```
>>> ma_liste = input_liste(5)
Entrez un nombre : 4
Entrez un nombre : 5
Entrez un nombre : 3
Entrez un nombre : 1
Entrez un nombre : 8
>>> ma_liste
[4, 5, 3, 1, 8]
```

Somme des éléments d'une liste (version 1)

```
1 def somme_liste_v1(l):
2     ''' ... '''
3     total = 0                    # initialise le total
4     for i in range(len(l)):      # pour i de 0 à len(l) - 1:
5         total += l[i]            #   somme l[i] à total
6     return total                 # retourne le total
```

```
>>> ma_liste = [5, 76, 8, 5, 3, 3, 56, 5, 23]
>>> somme_liste_v1(ma_liste)
184
```

Somme des éléments d'une liste (version 2)

```
1 def somme_liste_v2(l):
2     ''' ... '''
3     total = 0          # initialise le total
4     for e in l:        # pour chaque element e dans l:
5         total += e     #   somme e à total
6     return total       # retourne le total
```

```
>>> ma_liste = [5, 76, 8, 5, 3, 3, 56, 5, 23]
>>> somme_liste_v2(ma_liste)
184
```

Modification de listes

Multiplier par deux chaque élément de la liste :

```
1 def mult_deux(l):
2     for i in range(len(l)):    # pour i de 0 à len(l) - 1:
3         l[i] *= 2              #   multiplie l[i] par 2
```

```
>>> ma_liste = [5, 76, 8, 5, 3, 3, 56, 5, 23]
>>> mult_deux(ma_liste)
>>> ma_liste
[10, 152, 16, 10, 6, 6, 112, 10, 46]
```

Ceci ne fonctionne pas !

Pourquoi ce programme ne modifie pas la liste ?

```
1 def mult_deux_v2(l):
2     ''' ... '''
3     for e in l :     # pour chaque élément e dans l:
4         e *= 2       #   multiplie e par 2
```

```
>>> ma_liste = [5, 76, 8, 5, 3, 3, 56, 5, 23]
>>> mult_deux_v2(ma_liste)
>>> ma_liste
[5, 76, 8, 5, 3, 3, 56, 5, 23]
```

Parce que la variable e contient une **copie** de la valeur dans la liste.

Les chaînes et les tuples sont (presque) pareils !

```
>>> x = 'chaîne'
>>> y = (0, 1, 2, 3, 4, 5)
>>> x
'chaîne'
>>> y
(0, 1, 2, 3, 4, 5)
```

Accéder un éléments :

```
>>> x[0]
'c'
>>> y[0]
0
```

Accès par la droite :

```
>>> x[-1]
'e'
>>> y[-1]
5
```

Exercice

Problème :

Écrire la fonction `liste_10_10()` qui retourne une liste de 10 listes de 10 zéros :

```
[[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
```

Exercice

1. Créer une liste de 10 zéros :

```
def liste_10():
    l = []                      # l = liste vide
    for i in range(10):         # pour i de 0 à 9 :
        l.append(0)             #   ajoute 0 à l
    return liste                # retourne l
```

2. Faire la même chose 10 fois :

```
def liste_10_10():
    l = []                          # l = liste vide
    for i in range(10):             # pour i de 0 à 9 :
        l.append(liste_10())        #   ajoute liste_10() à l
    return l
```

Comment changer le 5e élément de la 5e liste ?

```
>>> l = liste_10_10()
>>> l[4][4] = 3
```

En effet, une liste de listes fonctionne comme un tableau !

Exercice

Problème :

Soit une constante contenant les abréviations des noms de mois définie comme suit :

```
MOIS = ['jan', 'fév', 'mar', 'avr', 'mai', 'jun', 'jui',
    'ago', 'sép', 'oct', 'nov', 'déc']
```

Écrivez la fonction `abrev_mois(n)` qui reçoit un nombre `n` en argument et retourne l'abréviation du `n`-ième mois de l'année. Retourner `None` si `n` n'est pas compris entre 1 et 12.

```
1 MOIS = ['jan', 'fév', 'mar', 'avr', 'mai', 'jun', 'jui',
      'ago', 'sép', 'oct', 'nov', 'déc']
2
3 def abrev_mois(n):
4     ''' ... '''
5     if 1 <= n <= 12:
6         return MOIS[n - 1]
```

Exercice

Problème :

Soit une constante contenant les abréviations des noms de mois définie comme suit :

```
MOIS = ['janfévmaravrmaijunjuiagoséoctnovdéc']
```

Écrivez la fonction `abrev_mois(n)` qui reçoit un nombre `n` en argument et retourne l'abréviation du `n`-ième mois de l'année. Retourner `None` si `n` n'est pas compris entre 1 et 12.

Exercice

Programme :

```
1  MOIS = 'janfévmaravrmaijunjuiagoséoctnovdéc'
2
3  def abrev_mois(n):
4      ''' ... '''
5      if not (1 <= n <= 12):          # si pas entre 1 et 12:
6          return None                 #   retourne None
7
8      abrev = ''                      # abrev = chaîne vide
9      pos = (n - 1) * 3               # ajuste la position
10     for i in range(3):              # pour i de 0 à 2:
11         abrev += MOIS[pos + i]      #   ajoute un caractère
12                                     #   de MOIS à abrev
13
14     return abrev                    # retourne abrev
```

Section 27

Dictionnaires

Dictionnaires

Un **dictionnaire** est une structure qui ressemble à une liste. Pourtant, au lieu des index, il contient des clés qui ne sont pas forcement numériques.

Syntaxe :

```
{ <cle1> : <val1>, <cle2> : <val2>, ..., <cleN> : <valN> }
```

Exemple :

```
>>> d = {'nom': 'DUPONT', 'prenom': 'Pierre', 'numero': 43}
>>> d['nom']
'DUPONT'
>>> d['prenom']
'Pierre'
>>> d['numero']
43
```

Opérations sur dictionnaires

La fonction `len` retourne la longueur :

```
>>> len(d)
3
```

La fonction `del` supprime un élément :

```
>>> d
{'nom': 'DUPONT', 'prenom': 'Patrick', 'numero': 43}
>>> del(d['numero'])
>>> d
{'nom': 'DUPONT', 'prenom': 'Patrick'}
```

Pour ajouter des éléments :

```
>>> d['age'] = 34
>>> d
{'nom': 'DUPONT', 'prenom': 'Patrick', 'age': 34}
```

Opérations sur dictionnaires

Attention : Il n'est pas possible d'adresser un élément d'un dictionnaire par sa position :

```
>>> d[0]
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    KeyError: 0
```

Opération sur dictionnaires

La méthode `keys` retourne les clés d'un dictionnaire :

```
>>> d.keys()
dict_keys(['nom', 'prenom', 'age'])
```

La méthode `values` retourne les valeurs d'un dictionnaire :

```
>>> d.values()
dict_values(['DUPONT', 'Pierre', 34])
```

La méthode `items` retourne les pairs formés par les clés et valeurs d'un dictionnaire :

```
dict_items([('nom', 'DUPONT'), ('prenom', 'Pierre'),
    ('age', 34)])
```

Section 28

Exemples avec dictionnaires

Itération sur les clés d'un dictionnaires

```python
def print_cles_v1(d):
    ''' ... '''
    for c in d :     # pour chaque clé c de d:
        print(c)     #   affiche c

def print_cles_v2(d):
    ''' ... '''
    for c in d.keys():  # pour chaque clé c de d:
        print(c)        #   affiche c
```

```
>>> d = {'a':1, 'b':2, 'c':3}
>>> print_cles_v1(d)
a
b
c
>>> print_cles_v2(d)
a
b
c
```

Itération sur les valeurs d'un dictionnaire

```
1 def print_vals_v1(d):
2     ''' ... '''
3     for c in d:           # pour chaque clé c de d:
4         print(d[c])       #   affiche d[c]
5
6 def print_vals_v2(d):
7     ''' ... '''
8     for v in d.values():  # pour chaque valeur v dans d:
9         print(v)          #   affiche v
```

```
>>> d = {'a':1, 'b':2, 'c':3}
>>> print_vals_v1(d)
1
2
3
>>> print_vals_v2(d)
1
2
3
```

Exercice

Problème : Soit une liste de dictionnaires où chaque entrée correspond à un étudiant d'une promotion :

```
promo = [ {'nom':'DUPONT', 'prenom':'Pierre', 'age':20},
          {'nom':'DULAC', 'prenom':'Jean', 'age':19},
          {'non':'DUMONT', 'prenom':'André', 'age':21},
          ... ]
```

Écrivez la fonction `age_plus_jeune(promo)` qui reçoit une promotion comme argument et retourne l'âge de l'étudiant le plus jeune de la promotion.

Exercice

Programme :

```
1 def age_plus_jeune(promo):
2     ''' ... '''
3     a = promo[0]['age']          # a = l'âge du premier
4
5     for etu in promo:            # pour chaque étudiant:
6         if etu['age'] < a:       #   si son âge < a:
7             a = etu['age']       #       a = son âge
8
9     return a                     # retourne a
```

Exercice

Problème : Écrivez la fonction `ages_de(promo, age)` qui reçoit une promotion et un âge comme arguments et retourne la liste des étudiants ayant cet âge.

Programme :

```
1 def ages_de(promo, age):
2     ''' ... '''
3     l = []                          # l = liste vide
4     for etu in promo:               # pour chaque étudiant:
5         if etu['age'] == age:       #   s'il a l'âge donnée:
6             l.append(etu)           #       ajoute-le à l
7
8     return l                        # retourne l
```

Exercice

Problème : Écrivez la fonction `les_plus_jeunes(promo)` qui reçoit une promotion comme argument et retourne la liste des étudiants les plus jeunes de la promotion.

Programme :

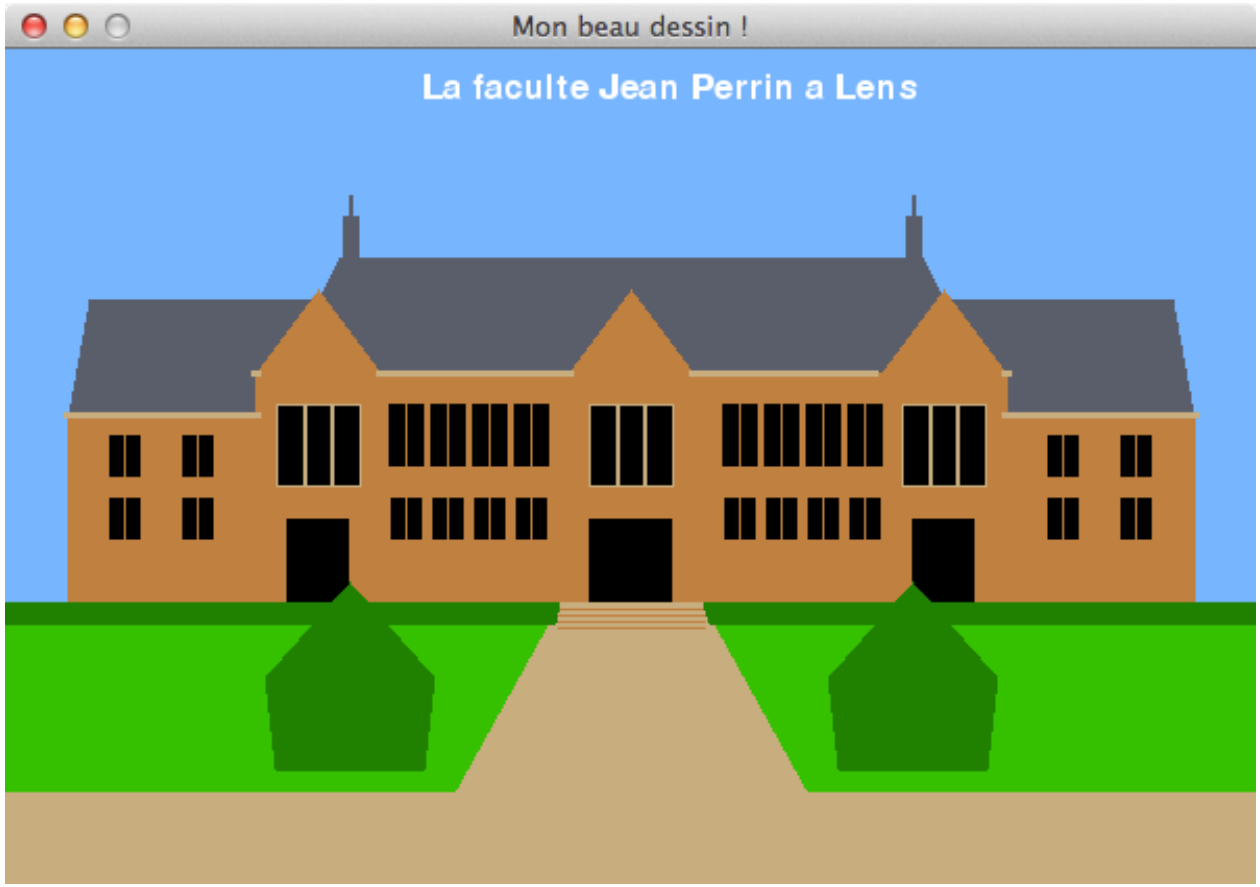
```
1 def les_plus_jeunes(promo):
2     ''' ... '''
3     a = age_plus_jeune(promo)      # a = âge du plus jeune
4     l = ages_de(promo, a)          # l = les étudiants
5                                    #     âgés de a
6     return l                       # retourne l
```

Cours 7

Graphismes

Aujourd'hui. . .

Nous allons faire un beau dessin !



La bibliothèque Pygame

Pour installer Pygame, visitez l'adresse : `http://pygame.org`

Pygame est une bibliothèque graphique parmi tant d'autres, comme tkinter, PySDL, PyQt, wxPython, etc.

Pygame est une bibliothèque créée, entre autres, pour faciliter :

- le dessin des formes graphiques basiques ;
- l'affichage des images type « bitmap » ;
- les animations ;
- le traitement du clavier, la souris, les manettes de jeu, etc. ;
- le traitement de son.

L'algorithme du programme

Un programme qui utilise Pygame pour afficher des objets graphiques doit suivre un schéma particulier pour pouvoir fonctionner correctement.

Que ce soit un jeu, une animation ou un simple dessin, le schéma est toujours le même.

1. Importe la bibliothèque Pygame.
2. Initialise Pygame.
3. Initialise l'horloge.
4. Ouvre la fenêtre de l'application.
5. Tant que le programme n'est pas terminé :
   - 5.1 Traite les événements.
   - 5.2 Ajuste la vitesse de la boucle.
   - 5.3 Dessine.
6. Termine le programme.

Nous allons illustrer l'utilisation de Pygame en utilisant le code source du dessin d'un simple rectangle comme exemple.

Initialisation du programme

Comme toute autre bibliothèque en Python, Pygame doit être importée avant son utilisation :

```
import pygame
```

Note : en effet, nous pouvons aussi utiliser « `from pygame import *` », mais ceci est déconseillé pour éviter que des fonctions soient redéfinies.

Ensuite, pour pouvoir utiliser les fonctionnalités de Pygame dans notre programme, nous devons l'initialiser.

```
pygame.init()
```

Initialisation de l'horloge

Dans l'étape suivante, nous devons initialiser l'horloge :

```
horloge = pygame.time.Clock()
```

La variable `horloge` contient maintenant une référence à l'horloge.

Ceci permettra l'exécution de l'étape « Ajuster la vitesse de la boucle », qui sera expliquée plus tard.

Ouverture de la fenêtre de l'application

La prochaine étape est l'ouverture de la fenêtre de notre application.

```
surface = pygame.display.set_mode((600, 400))
pygame.display.set_caption('Mon beau dessin !')
```

La première instruction ci dessus ouvre une fenêtre de taille 600×400 pixels pour l'application. **Les deux parenthèses sont nécessaires ! !**

La variable `surface` contient une référence à la surface de la fenêtre de l'application.

C'est sur cette `surface` que nous allons dessiner les objets.

La deuxième instruction ci dessus assigne un titre à la fenêtre de l'application.

La boucle principale du programme

La boucle principale du programme correspond à l'instruction :

« Tant que le programme n'est pas terminé : »

Cette boucle est nécessaire. Elle maintient ouverte la fenêtre de l'application jusqu'à ce que l'utilisateur demande sa fermeture.

Voici le code schématique de la boucle principale :

```
continuer = True
while continuer :

    # Traite les événements.

    # Ajuste la vitesse de la boucle.

    # Efface l'écran.

    # Dessine les objets.

    # Met à jours l'écran.
```

Terminer Pygame

Avant de terminer le programme, il faut terminer Pygame :

```
pygame.quit()
```

Cela peut éviter le blocage de certaines interfaces de Python, comme Thonny, par exemple.

Traitement des événements

C'est à l'intérieur de la boucle principale du programme que nous devons traiter les événements causés par l'utilisateur (clavier, souris, etc.).

Sans ce traitement, l'utilisateur ne peut pas interagir avec le programme. Il ne pourra même pas fermer la fenêtre du programme !

Le traitement des événements doit lui aussi être programmé comme une boucle :

1. Pour chaque événement `event` détecté :
   - 1.1 Si `event` est égal à « quitter » :
     - 1.1.1 Quitter.
   - 1.2 Si `event` est égal à « touche du clavier » :
     - 1.2.1 Faire quelque chose.
   - 1.3 Si `event` est égal à « button de la souris » :
     - 1.3.1 Faire quelque chose.
   - 1.4 ...

Traitement des événements

Code schématique du traitement des événements :

```
# Pour chaque événement detecté :
for event in pygame.event.get():

    # Si l'utilisateur a cliqué sur fermer
       fenêtre :
    if event.type == pygame.QUIT:
        terminer = True

    # Si une touche du clavier a ete appuyee :
    if event.type == pygame.KEYDOWN :
        # Remplir avec code qui traite cet é
           vénement.

    # Si le button de la souris a ete appuye :
    if event.type == pygame.MOUSEBUTTONDOWN :
        # Remplir avec code qui traite cet é
           vénement.
```

Ajuster la vitesse de la boucle

Avant de dessiner, nous devons demander au programme qu'il attende un peu.

C'est ici que nous utilisons la variable `horloge` qui à été initialisée dans l'étape « Initialisation de l'horloge ».

Voici la commande pour ajuster la vitesse de la boucle :

```
horloge.tick(40)
```

La valeur 40 passée en argument signifie que la boucle sera exécuté 40 fois par seconde (40 FPS).

Effacer l'écran

Avant de dessiner les objets, il est parfois utile d'effacer la surface de la fenêtre de l'application avec l'instruction `fill` :

```
surface.fill((0, 0, 0))
```

Attention : dans cet exemple, `surface` correspond à la variable définie auparavant avec l'instruction `pygame.display.set_mode`.

L'argument `(0, 0, 0)` correspond au code de la couleur du fond de l'écran. Dans ce cas, la couleur noir.

Il est possible de tinter l'écran avec n'importe quelle couleur. (Nous verons les codes de couleurs par la suite.)

Dessin des objets

Dans notre exemple, nous allons simplement dessiner un rectangle blanc. Voici donc l'instruction pour le faire :

```
pygame.draw.rect(surface, (255, 255, 255), (50,
    50, 200, 200))
```

Encore une fois, la variable `surface` correspond à la variable définie auparavant avec l'instruction `pygame.display.set_mode`.

Le tuple `(255, 255, 255)` correspond à la couleur du rectangle : blanc.

Le tuple `(50, 50, 200 200)` correspond, dans l'ordre :

- La coordonée `x` du rectangle `50`.
- La coordonée `y` du rectangle `50`.
- La largeur du rectangle `200`.
- La hauteur du rectangle `200`.

Mise à jours de l'écran

Après avoir dessiné sur la surface de l'écran, nous devons mettre à jours l'écran pour que les objets soient affichés :

```
pygame.display.update()
```

Attention : **il est nécessaire d'utiliser cette instruction pour que les objets soient affichés à l'écran.**

Cette instruction doit être exécutée une seule fois, à la fin de la boucle.

Le programme complet

Vérifiez le programme complet dans le fichier `affiche_rectangle.py`.

Couleurs

Avant de commencer à dessiner les objets, il est parfois utile de définir les couleurs qui seront utilisées.

Pygame utilise le système de couleurs RVB (rouge, vert, bleu), parfois aussi appelé RGB (de l'anglais : red, green, blue).

Chaque couleur est un mélange de ces trois couleurs représenté par un triplet d'entiers entre 0 et 255.

Par exemple :

- la couleur blanche est représentée par le code (255, 255, 255) ;
- la couleur noir est représentée par le code (0, 0, 0).

La valeur 0 signifie qu'il n'y a aucun pigment de la couleur correspondante dans le mélange alors que 255 signifie qu'il y a autant que possible du pigment correspondant dans le mélange.

Couleurs

Définitions de quelques couleurs au format RVB :

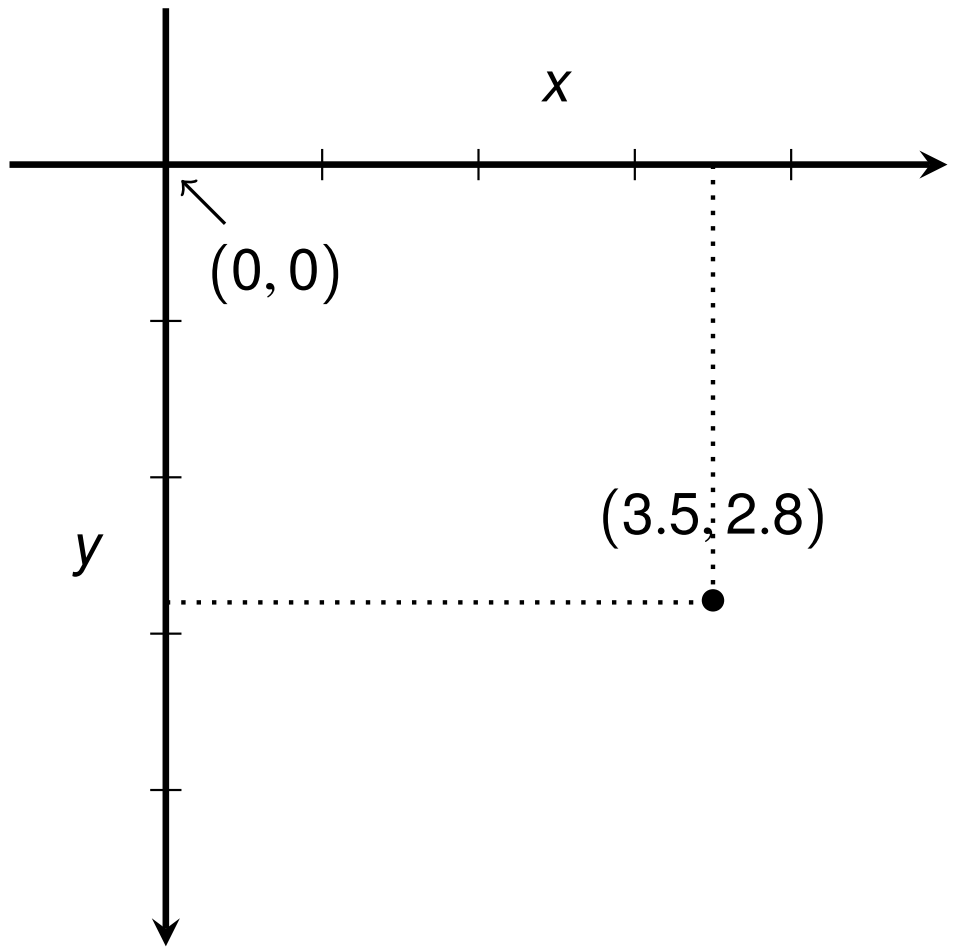
```
NOIR        = (  0,   0,   0)
BLANC       = (255, 255, 255)
BLEU_CIEL   = (119, 181, 254)
VERT        = (  0, 192,   0)
VERT_FONCE  = (  0, 128,   0)
BEIGE       = (200, 173, 127)
MARRON      = (192, 128,  64)
ARDOISE     = ( 90,  94, 107)
```

Pour trouver plus de couleurs au format RVB, regardez l'adresse :
`http://fr.wikipedia.org/wiki/Liste_de_noms_de_couleur`

Dessiner les objets

Pour dessiner des objets sur la surface de l'écran nous devons passer les coordonnées des objets que nous voulons dessiner.

Le système des coordonnées de l'ordinateur est semblable au système Cartésien, mais avec l'axe *y* inversé.



Dessiner un rectangle

Syntaxe :

```
pygame.draw.rect(Surface, couleur, coords, largeur=0)
```

`Surface` : surface où l'objet sera dessinée.
(Définie auparavant avec `pygame.display.set_mode`.)

`couleur` : couleur de l'objet.

`coords` : une séquence de quatre entiers. Les deux premiers correspondent aux coordonnées (x, y) de l'objet. Les deux autres à la largeur et à la hauteur de l'objet.

`largeur`: (paramètre optionnel, valeur par défaut = 0) un entier qui correspond à la largeur de la bordure de l'objet.

Exemple :

```
pygame.draw.rect(surface, BLANC, (50, 50, 200,
    200))
```

Dessiner une ligne

Syntaxe :

```
pygame.draw.line(Surface, couleur, debut, fin, largeur=1)
```

`Surface:` surface où l'objet sera dessiné.

`couleur:` couleur de l'objet.

`debut:` coordonnées du début de la ligne.

`fin:` coordonnées de la fin de la ligne.

`largeur:` (optionnel, défaut = 1) largeur de la ligne.

Exemple :

```
pygame.draw.line(surface, BLANC, (265, 268),
    (335, 268))
```

Dessiner d'autres objets

Il y a plusieurs instructions pour dessiner. Leur syntaxe est semblable.

Syntaxes :

```
pygame.draw.ellipse(Surf, couleur, coords, largeur=0)
```

```
pygame.draw.arc(Surf, couleur, debut, fin, largeur=0)
```

```
pygame.draw.polygon(Surf, couleur, points, largeur=0)
```

```
pygame.draw.circle(Surf, couleur, pos, rayon, largeur=0)
```

Pour plus d'objets, regarder :
`http://pygame.org/docs/ref/draw.html`

Dessiner du texte

L'écriture d'un texte en mode graphique consiste en trois étapes :

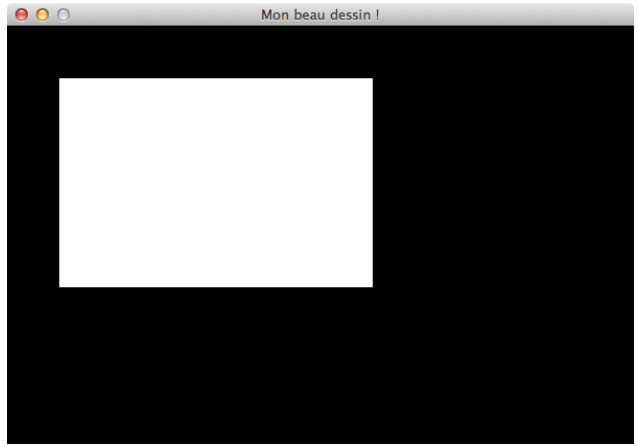
1. Sélection de la police.
2. Création de l'image du texte.
3. Affichage du texte.

Exemple :

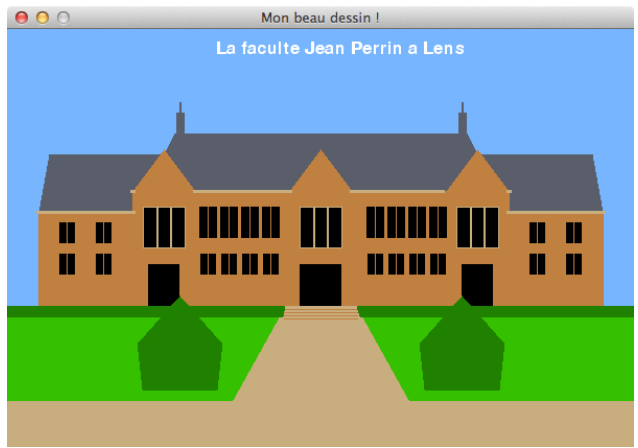
```
# Seléction de la police default et taille 25.
police = pygame.font.Font(None, 25)

# Création de l'image du texte.
texte = police.render('Mon texte', True, NOIR)

# Affichage du texte aux coordonnées (250, 250).
surface.blit(texte, (250, 250))
```

Fichiers source complets

- `affiche_rectangle.py`



- `affiche_jean_perrin/`
  - `__init__.py`
  - `__main__.py`
  - `jean_perrin.py`
  - `couleur.py`

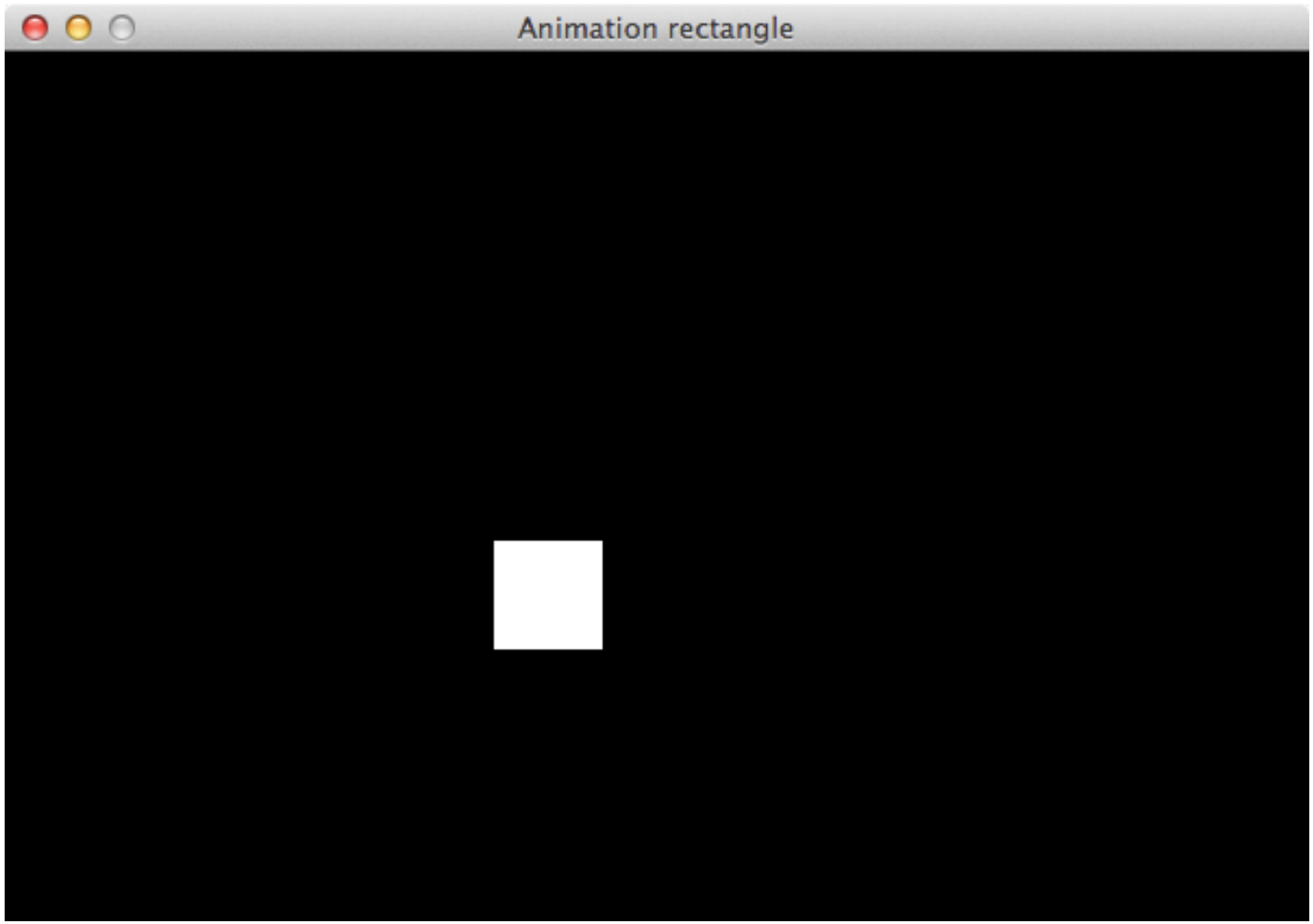


Cours 8

Animations

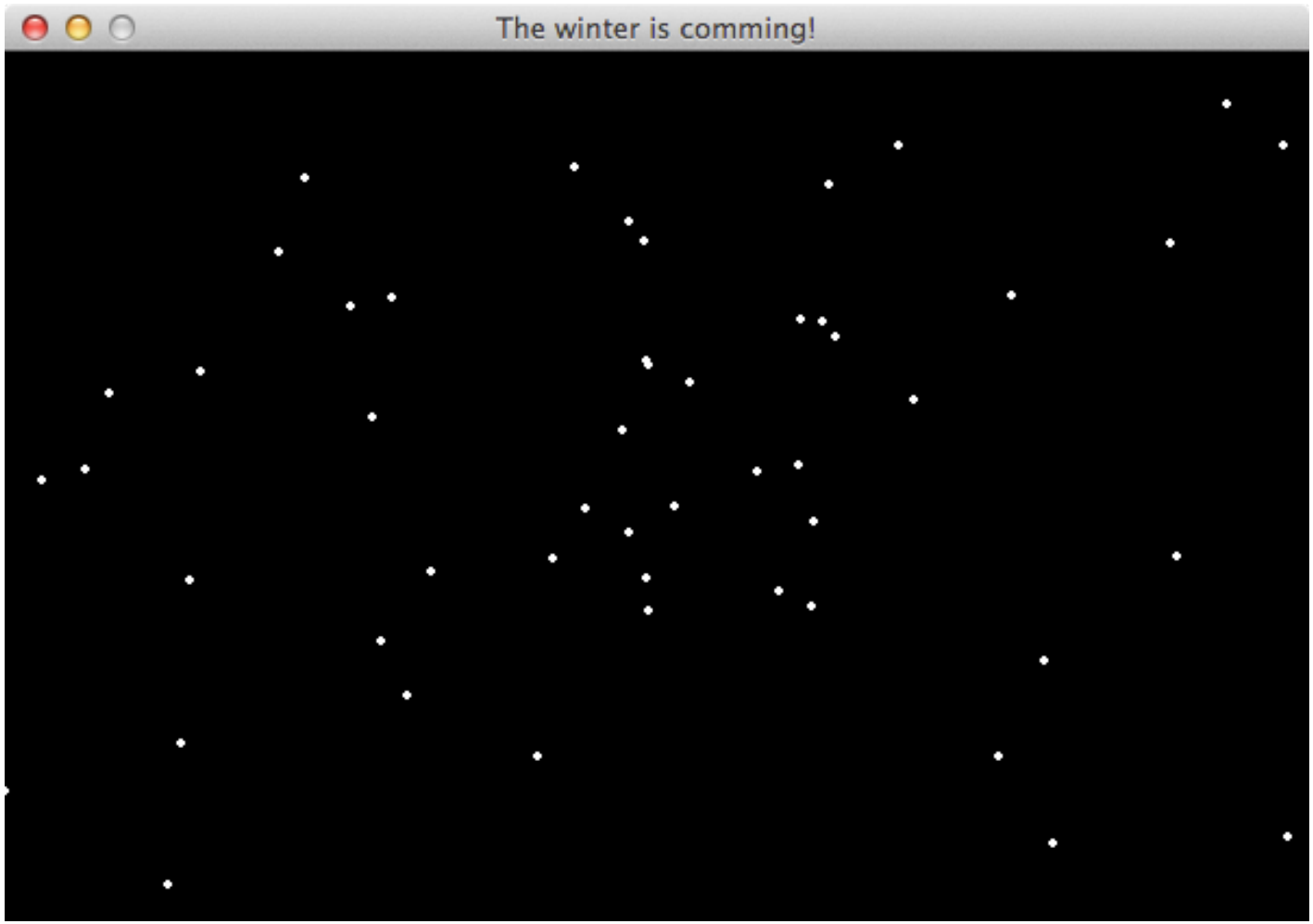
Aujourd'hui. . .

Nous allons faire un rectangle se balader sur l'écran !



Aujourd'hui. . .

Nous allons faire neiger !



Sommaire

Section 29

Schéma modulaire

Animation

Comme rappel, voici le schéma suivi par notre programme :

1. Importe la bibliothèque Pygame.
2. Initialise Pygame.
3. Initialise l'horloge.
4. Ouvre la fenêtre de l'application.
5. Tant que le programme n'est pas terminé :
   - 5.1 Traite les événements.
   - 5.2 Ajuste la vitesse de la boucle.
   - 5.3 Dessine les objets.
   - 5.4 Met à jour l'écran.
6. Termine Pygame.

Un programme animé n'est rien d'autre qu'un programme qui redessine constamment les objet à l'écran.

Donc, pour transformer ce schéma statique dans un schéma animé, nous devons faire un petit changement.

Animation

Voici donc le nouveau schéma (les changements sont en rouge) :

1. Importe la bibliothèque Pygame.
2. Initialise Pygame.
3. Initialise l'horloge.
4. Ouvre la fenêtre de l'application.
5. Initialise les objets de l'animation.
6. Tant que le programme n'est pas terminé :
   - 6.1 Traite les événements.
   - 6.2 Ajuste la vitesse de la boucle.
   - 6.3 Met à jour les objets.
   - 6.4 Dessine les objets.
   - 6.5 Met à jour l'écran.
7. Termine Pygame.

Un schéma modulaire

Nous allons utiliser un schéma modulaire pour nos programmes.

L'avantage du schéma modulaire est qu'il est plus facile de modifier.

Il sera plus facile de l'adapter pour créer d'autres animations plus tard.

Dans le répertoire `animation_rectangle`, nous allons créer 4 modules (c.à.d., 4 fichiers) :

- `__main__.py`
  contient la fonction principale.
- `anim.py`
  contient les fonctions spécifiques pour une animation.
- `couleur.py`
  contient les définitions de couleurs.
- `rectangle.py`
  contient les fonctions spécifiques au rectangle de l'animation.

Le module principal

Le fichier du module principal, appelé `__main__.py`, contient la fonction principale du programme et son appel.

La fonction `main` appelle trois fonctions du module `anim` :

- La fonction `init`, qu'initialise l'animation. Elle exécute toute les étapes qui précédent la boucle principale.
- La fonction `boucle` est responsable pour la boucle principale du programme.
- La fonction `quitte` est responsable pour terminer Pygame et quitter le programme.

Le module principal

Fichier __main__.py :

```
1  # Importe le module animation.
2  import anim
3
4  def main():
5      ''' Fonction principale. '''
6      # Initialise l'animation.
7      att_anim = anim.init()
8
9      # Exécute la boucle principale de l'animation.
10     anim.boucle(att_anim)
11
12     # Finalise l'animation.
13     anim.quitte()
14
15 # Appelle la fonction principale.
16 if __name__ == '__main__':
17     main()
```

Initialisation

L'initialisation contient les instructions que nous avons déjà vu auparavant :

- Importe la biblithèque Pygame.
- Initialise Pygame.
- Initialise l'horloge.
- Ouvre la fenêtre de l'application.

Les nouveautés sont :

- Importe les modules des objets de l'animation.
- Initialise les objets de l'animation.
  (Sera expliqué par la suite.)
- Crée le dictionnaire des attributs de l'animation.
  (Facilitera l'accès aux attributs dans les autres fonctions.)

Initialisation

Fichier anim.py :

```
1  # Importe la bibliothèque Pygame.
2  import pygame
3  # Importe les modules des objets de l'animation.
4  import rectangle
5  # Importe les couleurs
6  from couleur import NOIR
7
8  def init():
9      # Initialise Pygame.
10     pygame.init()
11
12     # Initialise l'horloge.
13     horloge = pygame.time.Clock()
14
15     # Ouvre la fenêtre de l'application.
16     fen_l = 600
17     fen_h = 400
```

Initialisation

Continuation de la fonction `init` du fichier `anim.py` :

```
20      # Initialise le(s) objet(s) de l'animation.
21      mon_rectangle = rectangle.init(50, 50, 65, 55,
22                                     fen_l, fen_h)
23
24      # Crée le dictionnaire des attributs de
           l'animation.
25      att = {}
26      att['horloge'] = horloge
27      att['surface'] = surface
28      # L'animation contient un seul objet, le
           rectangle.
29      att['rectangle'] = mon_rectangle
30      # Controle quand l'animation doit terminer.
31      att['tourner'] = True
32
33      # Retourne les attributs.
34      return att
```

Boucle principale du programme

La fonction `boucle` reçoit les attributs de l'animation.

Cette fonction a trois buts, chacun réalisé par une fonction différente :

- Traiter les événements.
- Mettre à jour les objets de l'animation.
- Dessiner les objets.

Fichier `anim.py` :

```
1 def boucle(att):
2     ''' Exécute la boucle principale de
          l'animation. '''
3     # Tant que le programme n'est pas terminé :
4     while att['tourner']:
5         traite_evenements(att)
6         met_a_jour(att)
7         dessine(att)
```

Terminer le programme

Pour l'instant, la fonction `quitte` doit uniquement terminer Pygame.

Fichier `anim.py` :

```
1 def quitte():
2     ''' Finalise le programme. '''
3     # Termine Pygame.
4     pygame.quit()
```

Traitement des événements

La fonction `traite_evenements` vérifie si l'utilisateur a décidé de quitter le programme. Si cela est le cas, elle assigne `False` à l'attribut `tourner`.

C'est ici que nous commençons a voir les avantages du schéma modulaire.

Notez que vous pouvez facilement ajouter le traitement d'autres evènements dans cette fonction.

Comme vos avez accès aux attributs de l'animation, et par conséquence, aux attributs des objets de l'animation, vous pouvez ajouter du code pour contrôler ces objets. (Nous allons faire cela plus tard.)

Traitement des evènements

Fichier anim.py :

```
1  def traite_evenements(att):
2      ''' Traite les evènements.
3      Arguments:
4          att : dict -- les attributs de
               l'animation.
5      '''
6       # Pour chaque evenement detecté :
7       for event in pygame.event.get():
8
9           # Si cliqué sur la croix rouge :
10          if event.type == pygame.QUIT :
11              att['tourner'] = False
12
13          # Si une touche est appuyée :
14          if event.type == pygame.KEYDOWN:
15              # Si la touche appuyée est ESC :
16              if event.key == pygame.K_ESCAPE:
```

Mise à jour

La fonction `met_a_jour` est responsable pour :

- Ajuster la vitesse de la boucle.
- Mettre à jour les objets.

Ici, nous avons un seul objet à mettre à jour, le rectangle.

Mais, dans une animation plus élaborée, nous pouvons avoir plusieurs objets à mettre à jour. Cette mise à jour sera faite ici.

Dans certains cas, l'ordre de mise à jour des objets est importante.

Mise à jour

Fichier : anim.py :

```
1  def met_a_jour(att):
2      ''' Met à jour l'animation.
3      Arguments:
4          att : dict -- les attributs de
               l'animation.
5      '''
6      # Ajuste la vitesse de la boucle.
7      # Tourne à 40 FPS.
8      att['horloge'].tick(40)
9
10     # Met à jour le(s) objet(s) de l'animation.
11     rectangle.met_a_jour(att['rectangle'],att['surfac
```

Dessiner les objets

La fonction `dessine` est responsable pour :

- Effacer l'écran.
- Dessiner tous les objets de l'animation.
- Mettre à jour l'écran.

Encore une fois, dans une animation plus élaborée, il peut y avoir plusieurs objets à dessiner.

L'ordre du dessin est importante, car les objets déssinés plus tard sont éventuelements mis par dessus des autres.

Dessiner les objets

Fichier anim.py :

```
1  def dessine(att):
2      ''' Dessine tous les éléments du jeu.
3      Arguments:
4          att : dict -- les attributs de
               l'animation.
5      '''
6      # Efface l'écran (tout noir).
7      att['surface'].fill(NOIR)
8
9      # Dessine le(s) objet(s).
10     rectangle.dessine(att['rectangle'],
           att['surface'])
11
12     # Met à jour l'écran.
13     pygame.display.update()
```

Le module rectangle

Dans notre schéma, les objets sont définis dans des modules séparés.

Dans le module de chaque objet, nous définissons les fonctions :

- `init` : initialise l'objet et ses attributs.
- `met_a_jour` : met à jour les attributs de l'objet.
- `dessine` : dessine l'objet sur la surface de l'application.

Le module rectangle

Nous avons l'intention de programmer le rectangle de sorte qu'à chaque exécution de la boucle principale, le rectangle se déplacé de 5 pixels vers la droite et vers le bas.

Donc, le rectangle sortira de l'écran si nous laissons l'animation tourner.

Nous faisons donc le rectangle changer de direction quand il atteint le bord de l'écran.

Pour cela, nous utilisons des variables pour la vitesse et la direction du rectangle.

Initialisation du rectangle

Fichier `rectangle.py` :

```
1  import pygame
2  from couleur import BLANC
3
4  def init(x, y, l, h, lim_x, lim_y):
5      ''' Initialise un rectangle dans les positions
6      pos_x et pos_y et de taille largeur, hauteur.
7      Enregistre également les positions limites.
8      Arguments : ...
9      Retour : ...
10     '''
11     att = {}
12     att['pos'] = [x, y]
13     att['taille'] = [l, h]
14     att['vitesse'] = [5, 5]
15     att['pos_lim'] = [lim_x - l, lim_y - h]
16     att['couleur'] = BLANC
17     return dict
```

Mise à jour du rectangle

La fonction `met_a_jour` est appelée à chaque tour dans la boucle principale de l'animation.

Cette fonction est responsable pour changer la position du rectangle.

D'abord, la fonction change la position du rectangle en utilisant sa vitesse de déplacement.

Ensuite, la fonction vérifie si le rectangle est sorti de l'écran. Si tel est le cas, la fonction repositionne le rectangle de façon à ce qu'il reste à l'écran.

Si le rectangle est au bord de l'écran, la fonction change aussi la direction du déplacement du rectangle.

Mise à jour du rectangle

Fichier `rectangle.py` :

```
1  def met_a_jour(rectangle):
2      ''' Change la position et/ou la direction du
           rect.
3      Arguments :
4          rectangle : dict -- les attributs d'un
               rectangle.
       '''
5
6      # Change la position du rectangle.
7      rectangle['pos'][0] += rectangle['vitesse'][0]
8      rectangle['pos'][1] += rectangle['vitesse'][1]
9
10     # Change la direction du rectangle si
           nécessaire.
11     if rectangle['pos'][0] >
           rectangle['pos_lim'][0] or \
12         rectangle['pos'][0] < 0 :
13          rectangle['vitesse'][0] *= -1
```

Dessin du rectangle

La fonction `dessine` est appelée à chaque tour dans la boucle principale du programme.

Elle est responsable pour dessiner le rectangle.

Fichier `rectangle.py` :

```
1 def dessine(rectangle, surface):
2     ''' Dessine un rectangle sur la surface.
3     Arguments:
4         rectangle : dict -- les attributs d'un
              rect.
5         surface : pygame.Surface -- la surface de
              l'app.
6     '''
7     pygame.draw.rect(surface, BLANC,
8         [rectangle['pos'][0], rectangle['pos'][1],
9          rectangle['taille'][0],
              rectangle['taille'][1]])
```

Section 30

Exemple de la neige

Les modules principal et animation

L'animation de la neige est faite de manière similaire à celle du rectangle.

Le module principal est le même que pour le rectangle, sans aucun changement.

Le module Animation (`anim.py`) est légèrement différent :

- Importation du module neige (au lieu du rectangle) :

```
import neige
```

- Initialisation de 50 flocons de neige dans la fonction `init` :

```
ma_neige = neige.init(50, fen_l, fen_h)
att['neige'] = ma_neige
```

- Mise à jour de la neige dans la fonction `met_a_jour` :

```
neige.met_a_jour(att['neige'])
```

- Dessin de la neige dans la fonction `dessine` :

```
neige.dessine(att['neige'])
```

Le module neige

La neige est une animation composée de plusieurs objets, ou flocons de neige. Donc, le module neige sera un peu différent du module rectangle.

Nous créons plusieurs flocons à la fois et de manière aléatoire à l'écran.

Le déplacement de tous les flocons est fait ensemble ainsi que son affichage.

La fonction `init` initialise la neige. Notamment :

- Définit la taille des flocons.
- Sauvegarde les coordonnées limites des flocons.
- Crée la liste de flocons de neige : chaque flocons est représenté par ses coordonnées `(x, y)` à l'écran.

Initialisation de la neige

Fichier : neige.py :

```
1  import pygame
2  from random import randrange
3  from couleur import BLANC
4
5  def init(n, lim_x, lim_y):
6      ''' Initialise n flocons de neige dans
7      les coordonnées limites passées en argument.
8      ...
9      '''
10     att = {}
11     att['taille'] = 2
12     att['couleur'] = BLANC
13     att['pos_lim'] = [lim_x, lim_y]
14     att['flocons'] = []
15     for i in range(n):
16         x = randint(0, lim_x)
17         y = randint(0, lim_y)
```

Mise à jour de la neige

La fonction `met_a_jour` est responsable pour la mise à jour de la neige.

Pour chaque flocon de neige :

- La coordonnée `y` du flocon est augmenté de 1 pixel. Donc, à la prochaine itération de la boucle, le flocon sera dessiné un peu plus bas à l'écran.
- Si le flocon sort de l'écran (c.à.d., la coordonnée `y` est supérieur à la position limite), la fonction remet le flocon en haut de l'écran. La nouvelle coordonnée `x` est choisi aléatoirement, pour que l'effet soit plus réaliste.

Mise à jour de la neige

Fichier `neige.py` :

```
1  def met_a_jour(neige):
2      ''' Met à jour la position des flocons de
           neige.
3      Arguments:
4          neige : dict -- les attributs de la neige.
5      '''
6      # Pour chaque flocon de neige
7      for i in range(len(neige['flocons'])):
8
9          # Déplace le flocon de neige d'un pixel
               vers
10         # le bas.
11         neige['flocons'][i][1] += 1
12
13         # Si le flocon de neige est arrivé en bas.
14         if neige['flocons'][i][1] >
               neige['pos_lim'][1]:
```

Dessin de la neige

La fonction `dessine` dessine tous le flocons de neige, ainsi que la neige au sol.

Pour chaque flocon de neige, la fonction fait appel à la fonction `pygame.draw.circle` pour dessiner le flocon correspondant à sa coordonnée `(x, y)`.

La neige déposée au sol est dessinée comme un rectangle blanc en bas de l'écran.

Dessin de la neige

Fichier `neige.py` :

```
1  def dessine(neige, surface):
2      ''' Dessine la neige.
3      Arguments :
4          neige : dict -- les attributs de la neige.
5          surface : pygame.Surface -- la surface
6      '''
7      # Pour chaque flocon de neige :
8      for n in neige['flocons']:
9          # Dessine.
10         pygame.draw.circle(surface, BLANC, n,
11                            neige['taille'])
12
13     # Dessine la neige au sol.
14     pygame.draw.rect(surface, BLANC,
15                      (0, neige['pos_lim'][1] - 10,
16                       neige['pos_lim'][0], 10) )
```

Section 31

Faire une pause

Faire une pause

Maintenant, nous désirons assigner à la touche « P » du clavier, la fonctionnalité de « pause ».

C'est-à-dire, nous voulons que quand l'utilisateur appuie une première fois sur la touche P, l'animation se met en pause. Quand l'utilisateur appuie une deuxième fois sur cette touche, l'animation reprend.

Nous désirons également que le mot « Pause » s'affiche quand l'animation est en pause.

Le code de la touche P du clavier est « `pygame.K_p` ».

À votre avis, quelles sont les fonctions et modules qui doivent être changées ?

Faire une pause

En effet, nous devons faire quatre changements :

- Nous allons créer un nouvel attribut pour indiquer que l'animation est en pause. Cet attribut, appelé `pause` sera initialisé à `False` dans la fonction `init`.
- L'attribut `pause` sera mis à `True` quand l'utilisateur appuie sur « P » la première fois et sur `False` si l'utilisateur re-appuie. Donc, nous changeons aussi le traitement des événements pour prendre en compte cette touche.
- Dans la fonction de mise à jour de l'animation, nous devons vérifier l'attribut `pause`. Si la valeur de l'attribut est `True`, alors nous ne faisons pas la mise à jour des objets. Comme cela, les objets seront figés à l'écran.
- Finalement, dans la fonction `dessine`, nous vérifions encore une fois l'attribut `pause`. S'il est `True`, alors nous affichons le mot « Pause » à l'écran.

Initialisation

Création de l'attribut `pause` dans la fonction `init` :

```
def init():
    ''' Initialise l'animation.
    Retour :
        dict -- les attributs d'une animation.
    '''
    ...

    att['pause'] = False

    ...
```

Traitement des événements

Traitement du nouvel événement (la touche « P ») :

```
1  def traite_evenements(att):
2      ''' Traite les événements.
3          ...
4      '''
5      # Pour chaque événement détecté :
6      for event in pygame.event.get():
7          # Si cliqué sur la croix rouge :
8          if event.type == pygame.QUIT:
9              att['tourne'] = False
10
11         # Si une touche a été appuyée :
12         if event.type == pygame.KEYDOWN:
13             # Si la touche appuyée est ESC:
14             if event.key == pygame.K_ESCAPE:
15                 att['tourne'] = False
16
17             # Si la touche appuyée est P:
```

Mise à jour

Si l'animation est en pause, alors les objets ne sont pas mis à jour :

```
1  def met_a_jour(att):
2      ''' Met à jour les objets.
3      Arguments :
4          att : dict -- les attributs d'une
               animation.
5      '''
6      # Ajuste la vitesse de la boucle (40 FPS).
7      att['horloge'].tick(40)
8
9      # Si l'animation n'est pas en pause:
10     if not att['pause']:
11         # Met à jours les objets.
12         neige.met_a_jour(att['neige'])
```

Dessin

Si l'animation est en pause, alors affiche « Pause », à l'écran :

```
1  def dessine(att):
2      ''' Dessine les objets de l'animation.
3      Arguments :
4          att : dict -- les attributs d'une
               animation.
5      '''
6      # Efface l'écran.
7      att['surface'].fill(NOIR)
8
9      # Dessine les objets.
10     neige.dessine(att['neige'], att['surface'])
11
12     # Si en pause:
13     if att['pause']:
14         affiche_pause(att)
15
16     # Met à jours l'écran.
```

Affichage de la pause

```
1  def affiche_pause(att):
2      ''' Affiche une pause.
3          ...
4      '''
5      # Charge la police.
6      pause_pol = pygame.font.Font(None, 50)
7
8      # Crée l'image du texte du mot "Pause".
9      pause_txt = pause_pol.render('Pause', True,
           NOIR, BLANC)
10
11     # Affiche "Pause" au milieu de l'écran.
12     att['surface'].blit(
13         pause_txt,
14         ( att['fen_l'] // 2 -
              pause_txt.get_width() // 2,
15          att['fen_h'] // 2 -
                pause_txt.get_height() // 2 )
```

Cours 9

Jeux

Aujourd'hui . . .

nous allons faire un Pong !

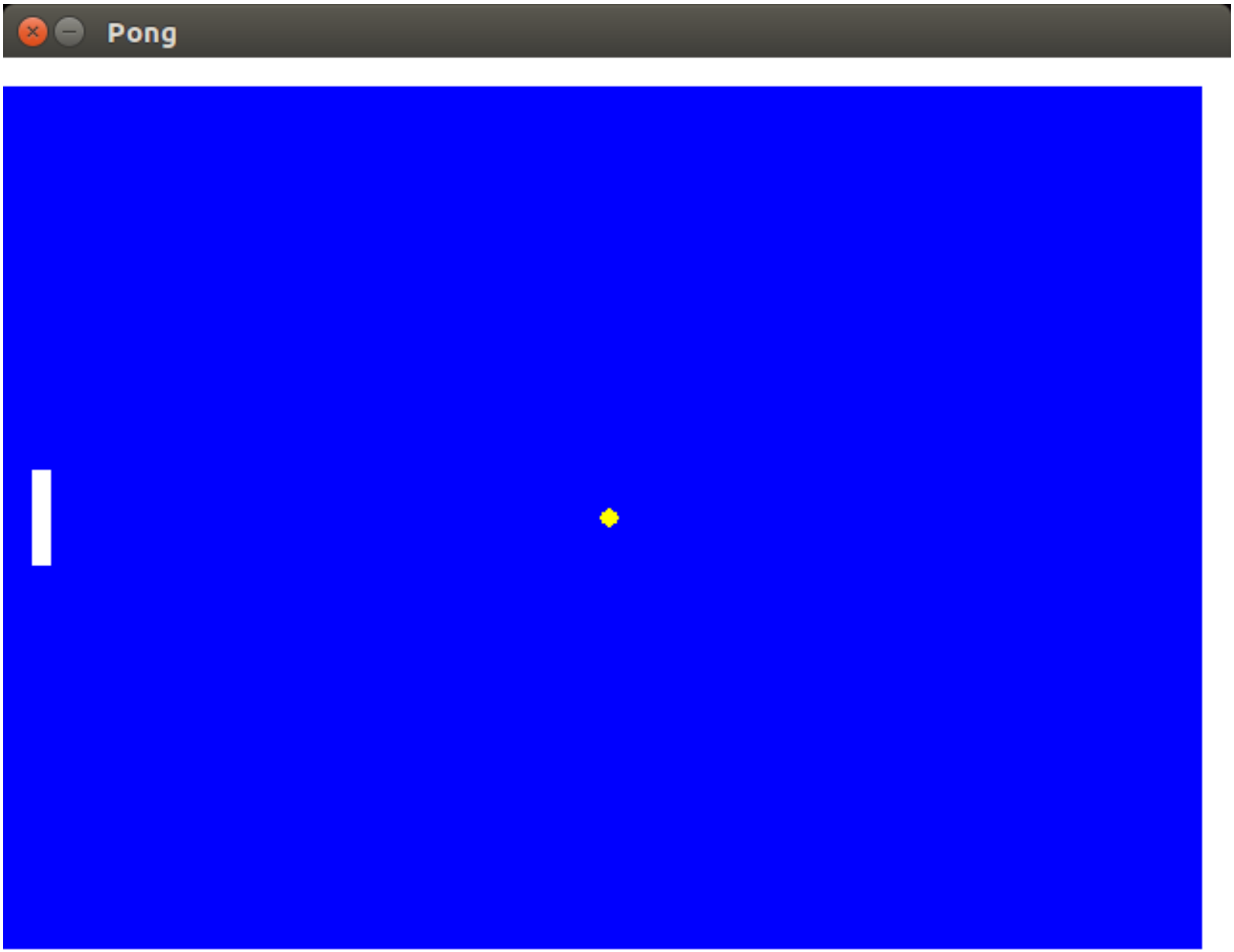


Schéma modulaire

Comme rappel, voici le schéma modulaire que nous avons construit pour l'animation du rectangle :

- `__main__.py`
  - `main()` (la fonction principale)
- `couleur.py`
- `anim.py`
  - `init()` (initialise Pygame, la fenêtre, l'horloge et les attributs)
  - `boucle()` (exécute la boucle principale)
  - `quittte()` (quitte Pygame)
  - `traite_evenements()` (traite les événements quitte, pause, etc.)
  - `met_a_jour()` (met à jour les objets de l'animation)
  - `dessine()` (efface l'écran, dessine les objets et met à jour l'écran)
- `rectangle.py`
  - `init()` (initialise les attributs du rectangle)
  - `met_a_jour()` (met à jour la position du rectangle)
  - `dessine()` (dessine le rectangle)

Schéma modulaire du jeu

Pour réaliser le jeu Pong, nous allons utiliser un schéma similaire.

- `__main__.py`
- `couleur.py`
- `jeu.py` (remplace `anim.py`)
- `terrain.py` (objet terrain du jeu)
- `balle.py` (objet balle)
- `palette.py` (objet palette)

Module principal

Le seul changement dans le module principal (`__main__.py`) est l'utilisation du module `jeu` au lieu du module `anim`.

```
1  # Importe les fonctions spécifiques du jeu.
2  import jeu
3
4  def main():
5      ''' Fonction principale. '''
6      # Initialise le jeu.
7      jeu_att = jeu.init()
8
9      # Exécute la boucle principale du jeu.
10     jeu.boucle( jeu_att )
11
12     # Termine le jeu.
13     jeu.quitte()
14
15 # Appelle la fonction principale.
16 if __name__ == '__main__':
```

Module jeu

Le module `jeu.py` doit :

- importer les objets du jeu ;
- importer les couleurs ;
- définir les types d'événements à traiter.
- définir les fonctions (les mêmes que pour le module `anim.py`) :
  - `init()`
  - `boucle()`
  - `quitte()`
  - `traite_evenements()`
  - `met_a_jour()`
  - `dessine()`
  - `affiche_pause()`

Module jeu

Le début du module jeu :

```
1 # Importe les objets du jeu.
2 import terrain, balle, palette
3
4 # Importe les couleurs.
5 from couleur import BLANC, BLEU, NOIR
```

Module jeu

Par la suite, le module jeu implémente les mêmes fonctions que le module `anim.py`.

Les fonctions `boucle()`, `quitte()` et `affiche_pause()` restent les mêmes que pour l'animation.

Les autres fonctions doivent être adaptées pour le jeu.

Initialisation du jeu

L'initialisation du jeu est faite quasiment de la même façon que pour l'animation.

La seule différence est la création des objets du jeu :

```
6  def init():
7      ''' Initialise le jeu et ses attributs.
8      Retour :
9          dict -- les attributs du jeu.
10     '''
11     ...
12
13     # Crée les objets du jeu.
14     jeu['terrain'] = terrain.init(jeu['fen_dim'])
15     jeu['balle'] = balle.init(jeu['fen_dim'])
16     jeu['palette'] = palette.init(jeu['fen_dim'])
17
18     ...
```

Traitement des événements

La fonction `traite_evenements()` subit des changements importants car nous devons maintenant traiter d'autres touches du clavier.

```
19 def traite_evenements(jeu):
20     ''' ... '''
21     # Pour chaque événement détecté :
22     for event in pygame.event.get():
23 
24         # Si cliqué sur la croix rouge :
25         ...
26         # Si une touche est appuyée :
27         if event.type == pygame.KEYDOWN:
28 
29             # ESC (quitte) :
30             ...
31             # P (pause) :
32             ...
```

Traitement des événements

```
33              # Haut (bouge la palette):
34              if event.key == pygame.K_UP:
35                  palette.haut(jeu['palette'])
36
37              # Bas (bouge la palette) :
38              if event.key == pygame.K_DOWN:
39                  palette.bas(jeu['palette'])
40
41          # Si une touche est relachée :
42          if event.type == pygame.KEYUP:
43
44              # Haut ou bas (arrête de bouger la
                    palette) :
45              if event.key == pygame.K_UP or \
46                  event.key == pygame.K_DOWN:
47
48                  palette.arrete(jeu['palette'])
```

Mise à jour du jeu

La fonction de mise à jour met à jour les objets du jeu et vérifie si le jeu est fini.

```
49 def met_a_jour(jeu):
50     ''' ... '''
51     # Ajuste la vitesse de la boucle.
52     # (Tourne le jeu à 40 FPS.)
53     jeu['horloge'].tick(40)
54 
55     # Si pas en pause :
56     if not jeu['pause']:
57         # Mets à jour les objets du jeu.
58         palette.met_a_jour(jeu['palette'])
59         balle.met_a_jour(jeu['balle'],
               jeu['palette'])
60 
61         # Vérifie si le joueur a perdu.
62         if jeu['balle']['pos'][0] < \
63             jeu['palette']['pos'][0]:
```

Dessin du jeu

La fonction qui dessine l'écran du jeu dessine les objets du jeu :

```
65 def dessine(jeu):
66     ''' ... '''
67     # Efface l'écran.
68     jeu['surface'].fill(NOIR)
69
70     # Dessine les objets du jeu.
71     terrain.dessine(jeu['terrain'],
           jeu['surface'])
72     balle.dessine(jeu['balle'], jeu['surface'])
73     palette.dessine(jeu['palette'],
           jeu['surface'])
74
75     # Si le jeu est en pause :
76     if jeu['pause']:
77         affiche_pause(jeu)
78
79     # Met à jour l'écran.
```

Module terrain

Le module terrain (`terrain.py`) implémente le terrain du jeu.

Le terrain du jeu ne change pas d'apparence ni de position.

C'est pourquoi ce module n'a pas besoin d'une fonction de mise à jour.

Donc, les fonctions à définir sont :

- `init()` (initialise les attributs d'un terrain)
- `dessine()` (dessine un terrain)

Tout d'abord, notez que nous avons besoin des dans ce module fonction de la bibliothèque Pygame pour dessiner ainsi que certaines couleurs.

Donc, la première chose à faire dans ce module sont les importations :

```
1 import pygame
2 from couleur import BLANC, BLEU
```

Initialisation du terrain

La fonction `init()` initialise les attributs du terrain :

```
3  def init(dim):
4      ''' Crée et initialise un terrain de jeu.
5      Arguments :
6          dim : (int, int) -- dimensions fenêtre du
               jeu.
7      Retour :
8          dict -- les attributs du terrain.
9      '''
10     (l, h) = dim
11     terrain = {
12         'couleur_fond' : BLEU,
13         'couleur_mur' : BLANC,
14         'rect_fond' : (0, 0, l, h),
15         'rect_haut' : (0, 0, l, 15),
16         'rect_droite' : (l - 15, 0, 15, h),
17         'rect_bas' : (0, h - 15, l, 15)
18     }
```

Dessin du terrain

La fonction dessine() dessine le terrain du jeu :

```
20 def dessine(terrain, surface):
21     ''' Dessine le terrain du jeu.
22     Arguments:
23         terrain : dict -- les attriburs d'un
               terrain.
24         surface : Surface -- la surface de
               l'applic.
25     '''
26     pygame.draw.rect(surface,
           terrain['couleur_fond'],
27                      terrain['rect_fond'])
28     pygame.draw.rect(surface,
           terrain['couleur_mur'],
29                      terrain['rect_haut'])
30     pygame.draw.rect(surface,
           terrain['couleur_mur'],
31                      terrain['rect_droite'])
```

Module balle

Le module balle (`balle.py`) implémente la balle jaune.

La balle doit rebondir sur les murs du terrain. Ceci ressemble beaucoup le rectangle de l'animation faite auparavant.

Les fonctions à définir sont :

- `init()` (initialise les attributs d'une balle)
- `met_a_jour()` (met à jour la position d'une balle)
- `dessine()` (dessine une balle)

Comme pour le terrain, nous avons besoin dans ce module des fonction de Pygame pour dessiner ainsi que de la couleur jaune :

```
1 import pygame
2 from couleur import JAUNE
```

Initialisation de la balle

La fonction `init()` initialise les attributs de la balle :

```
3  def init(dim):
4      ''' Crée et initialise une balle.
5      Argument :
6          dim : (int, int) -- dimensions de la
               fenêtre
7      Retour :
8          dict -- les attributs de la balle.
9      '''
10     (l, h) = dim
11     balle = {
12         'couleur' : JAUNE,
13         'lim' : (0, 15, l - 15, h - 15),
14         # Poisition du milieu de la balle.
15         'pos' : [l // 2, h // 2],
16         'rayon' : 5,
17         'vit' : [5, 5]
18     }
```

Mise à jour de la balle

La fonction `met_a_jour()` change la position de la balle, mais aussi, fait la balle rebondir sur les murs et sur la palette.

```
20 def met_a_jour(balle, palette):
21     ''' Met à jour la position d'une balle.
22     Arguments :
23         balle : dict -- les attributs d'une balle.
24         palette : dict -- les attributs d'une
               palette.
25     '''
26     # Change la position de la balle.
27     balle['pos'][0] += balle['vit'][0]
28     balle['pos'][1] += balle['vit'][1]
```

Mise à jour de la balle

```
29      # Raccourcis.
30      bx = balle['pos'][0]
31      by = balle['pos'][1]
32      br = balle['rayon']
33      tx1 = balle['lim'][0]
34      ty1 = balle['lim'][1]
35      tx2 = balle['lim'][2]
36      ty2 = balle['lim'][3]
37
38      # Si la balle sort du terrain (gauche ou
            droite) :
39      if (bx - br < tx1) or (bx + br > tx2):
40          # Inverse la vitesse horizontale de la
                balle.
41          balle['vit'][0] *= -1
42
43      # Si la balle sort du terrain (haut ou bas) :
44      if (by - br < ty1) or (by + br > ty2):
```

Mise à jour de la balle

```
47      # Raccourcis.
48      px = palette['pos'][0]
49      py = palette['pos'][1]
50      pl = palette['dim'][0]
51      ph = palette['dim'][1]
52
53      # Si la balle touche la palette:
54      if (bx - br < px + pl) and \
55          (py - br <= by <= py + ph + br):
56          # Inverse la vitesse horizontale de la
                balle.
57          balle['vit'][0] *= -1
```

Dessin de la balle

La fonction `dessine()` dessine la balle :

```
58 def dessine(balle, surface):
59     ''' Dessine une balle dans le jeu.
60     Argument :
61         balle : dict -- les attributs d'une balle.
62         surface : Surface -- la surface de
               l'applic.
63     '''
64     pygame.draw.circle(surface, balle['couleur'],
65                        balle['pos'],
                              balle['rayon'])
```

Module palette

Le module palette (`palette.py`) implémente la palette du jeu.

La palette bouge conformement l'utilisateur utilise les touches du clavier.

Donc, nous avons quelques fonctions de plus à définir :

- `init()` (initialise les attributs d'une palette)
- `arrete()` (arrête de bouger une palette)
- `bas()` (bouge une palette vers le bas)
- `haut()` (bouge la palette vers le haut)
- `met_a_jour()` (met à jour une palette)
- `dessine()` (dessine une palette)

Encore une fois, nous avons besoin des fonctions de Pygame pour dessiner ainsi que la couleur blanche :

```
1 import pygame
2 from couleurs import BLANC
```

Initialisation de la palette

La fonction `init()` initialise les attributs d'une palette :

```
3  def init(dim):
4      ''' Crée et initialise une palette
5      Arguments :
6          dim : (int, int) -- dimensions de la
               fenêtre.
7      Retour :
8          dict -- les attributs de la palette.
9      '''
10     (l, h) = dim
11     palette = {
12         'couleur' : BLANC,
13         'dim' : (10, 50),
14         'dir' : 0,
15         'lim' : (15, h - 50 - 15),
16         'pos' : [15, h // 2 - 25],
17         'vit' : 10
18     }
```

Mouvement de la palette

Les fonctions `arrete()`, `bas()` et `haut()` servent à contrôler le mouvement de la palette.

Ces fonctions sont appellées par le module jeu quand l'utilisateur utilise le clavier.

```
20 def arrete(pal):
21     ''' ... '''
22     pal['dir'] = 0
23
24 def bas(pal):
25     ''' ... '''
26     pal['dir'] = 1
27
28 def haut(pal):
29     ''' ... '''
30     pal['dir'] = -1
```

Mise à jour de la palette

La fonction `met_a_jour()` met à jour la position de la palette :

```
31 def met_a_jour(palette):
32     ''' Met à jour une palette.
33     Arguments :
34         palette : dict -- les attributs d'une
               palette
35     '''
36     # Change la position (uniquement verticale).
37     palette['pos'][1] += palette['vit'] *
           palette['dir']
38
39     # Si la palette sort du terrain (haut).
40     if( palette['pos'][1] < palette['lim'][0] ):
41         # Fige la position de la palette.
42         palette['pos'][1] = palette['lim'][0]
43
44     # Si la palette sort du terrain (bas).
45     if( palette['pos'][1] > palette['lim'][1] ):
```

Dessin de la palette

La fonction `dessine()` dessine la palette.

```
48 def dessine(palette, surface):
49     ''' Dessine une palette dans le jeu.
50     Argument :
51         palette : dict -- les attributs d'une
               palette
52     '''
53     rect = (palette['pos'][0],
54             palette['pos'][1],
55             palette['dim'][0],
56             palette['dim'][1])
57     pygame.draw.rect(surface, palette['couleur'],
          rect)
```