Utilisation de modules

Contenus	Capacités attendues	Commentaires
Utilisation de bibliothèques	Utiliser la documentation d'une bibliothèque	Aucune connaissance exhaustive d'une bibliothèque particulière n'est à connaître

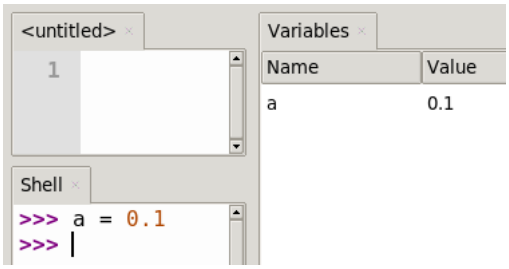
Il est possible de rassembler dans un même fichier des fonctions que l'on a écrites pour être utilisées dans un programme. Ce programme principal ('main' en anglais) doit alors importer ces fonctions pour pouvoir les utiliser.

Introduction aux espaces de noms

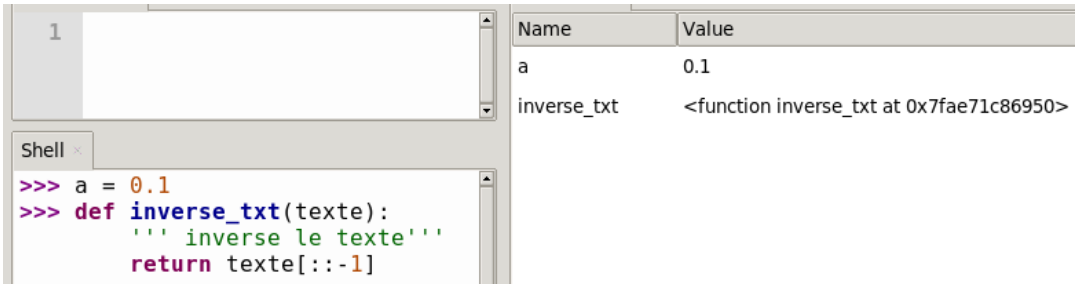
Ouvrir Thonny pour travailler dans l'interpréteur. Vérifier que la fenêtre 'Variables' soit bien affichée à droite, sinon aller dans le menu 'View' et sélectionner 'Variables'

On définit ici une variable : **>>> a = 0.1**

Après validation, la fenêtre 'Variables' fait apparaître le nom et la valeur associée :

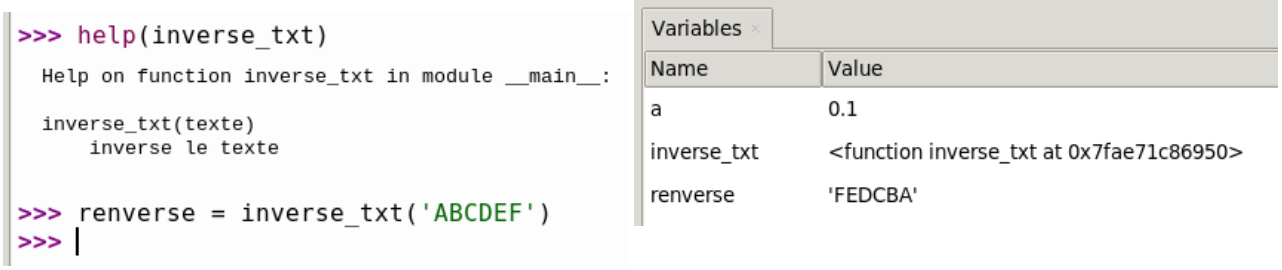


Définissons maintenant une fonction :



Cette fonction est maintenant connue : elle fait partie au même titre que la variable 'a' de l'**espace de noms** courant. Il est alors possible de :

- demander de l'aide sur cette fonction **>>> help(inverse_txt)**
- de l'appeler : **renverse = inverse_txt('ABCDEF')**



La variable 'renverse' fait maintenant partie elle aussi de l'espace de noms courant. On peut donc l'utiliser, par exemple en appelant la fonction inverse_txt dessus :

Remarques :

1. la fonction a bien réalisé le renversement
2. la variable ‘renverse’ n’a pas été modifiée (elle est de type str, un type qui n’est pas mutable)

```
>>> inverse_txt(renverse)
'ABCDEF'

>>> renverse
'FEDCBA'
```

Variables

Name	Value
a	0.1
inverse_txt	<function inverse_txt at 0x7fae71c86950>
renverse	'FEDCBA'

On voudrait maintenant calculer le sinus de a :

>>> **sin(a)**

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

L’erreur est claire : ‘sin’ n’est pas défini.

En effet, Python n’a pas dans son langage la fonction sinus. Par contre elle est présente dans le module ‘math’. Il faut donc importer ce module :

>>> import math

...qui apparaît dans l’espace de noms,

Variables

Name	Value
a	0.1
inverse_txt	<function inverse_txt at 0x7fae71c86950>
math	<module 'math' (built-in)>
renverse	'FEDCBA'

On peut alors essayer un :

>>> **help(math)** pour avoir l’aide sur ce module ou

>>> **dir(math)** pour afficher son contenu :

['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'remainder', '**sin**', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']

On y trouve notamment des fonctions (comme la fonction sinus : ‘sin’ , mais aussi des constantes comme ‘pi’)

Essayons alors :

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

>>> **sin(a)**

mais le problème reste le même : ‘sin’ n’est pas connu…

En fait, il faut comprendre que cette fonction sinus se trouve dans l’espace de noms du module math et non pas dans l’espace de noms courant. Le ‘cloisonnement’ entre ces deux espaces empêche l’interpréteur Python d’associer le mot ‘sin’ à la fonction sinus du module math.

Il faut alors préciser à l’interpréteur d’aller chercher dans l’espace de noms du module math. Cela se fait au moyen de la notation pointée : math.sin( )

>>> **math.sin(a)**

0.09983341664682815

Pour demander l’aide sur la fonction sinus :

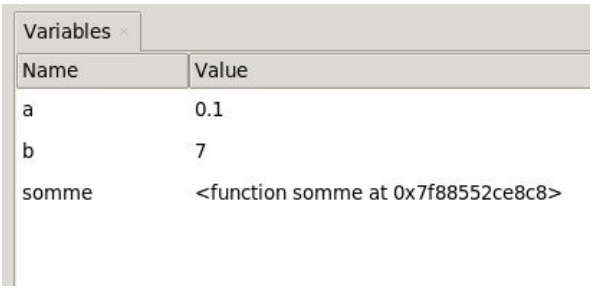
>>> **help(math.sin)**

```
>>> help(math.sin)
  Help on built-in function sin in module math:

  sin(x, /)
      Return the sine of x (measured in radians).
```

Arrêter l'interpréteur (icône Stop) et effacer son contenu pour effectuer la suite.

Ecrivons ces quelques lignes :

```
>>> a = 0.1
>>> def somme(a, b):
    s = a + b
    return s

>>> b = somme(3,4)
>>>
```



On remarque que le nom de la variable 's' n'apparaît pas dans l'espace de noms courant : **une fonction possède son propre espace de noms. Toute variable déclarée dans une fonction sera invisible en dehors de la fonction :**

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

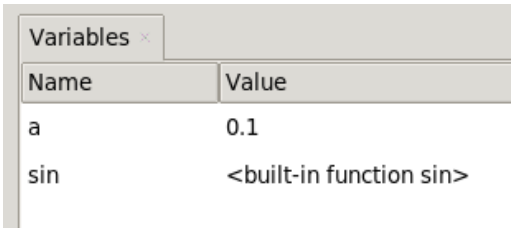
On a rencontré (notamment avec la carte microbit) une autre façon d'importer des fonctions :

from microbit import *

Dans cette expression, le symbole étoile signifie 'tout'

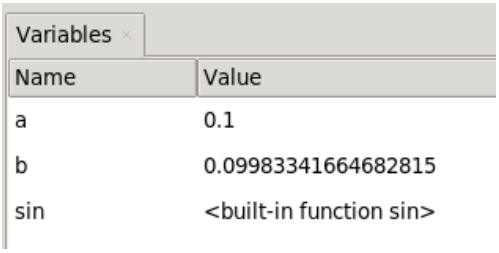
Essayons ce mode d'importation dans l'interpréteur Thonny, en décidant d'importer uniquement la fonction sinus :

```
>>> a = 0.1
>>> from math import sin
>>>
```



sin apparaît dans l'espace de noms courant. On peut donc l'utiliser tel quel :

```
>>> help(sin)
Help on built-in function sin in module math:

sin(x, /)

    Return the sine of x (measured in radians).

>>> b = sin(a)
```



Essayer une importation avec le joker ('wild card' en anglais) * :

>>> **from math import *** … et observer l'onglet des variables !

Quel type d'import préférer ?

Le tableau suivant résume les deux types d'importation réalisables en comparant avantages et inconvénients :

import *nom_du_module*	from *nom_du_module* import *nom_fonction*
- on importe le module : seul le nom du module apparaît dans l'espace de noms courant - plus lourd à l'utilisation car il faut utiliser la notation pointée pour utiliser la fonction voulue : *nom_du_module.nom_fonction* - mais plus sûre grâce à la qualification du nom fournie par la notation pointée on sait que la fonction utilisée est bien celle du module en question - possibilité d'aliasing pour alléger l'écriture (voir ci-dessous)	- on importe une ou plusieurs (voire toutes – avec l'option *) les fonctions du module dans l'espace de noms courant... - syntaxe plus légère : on utilise directement le nom de la fonction - risque d'écrasement de la fonction par une fonction de même nom redéfinie dans le code ou présente dans un autre module importé de la même façon (que va-t'il se passer si une autre fonction 'sin' est définie après l'importation du module math ?
A préférer systématiquement	**A réserver à des projets spécifiques n'utilisant pas d'autres modules**

aliasing : la méthode d'importation recommandée (import *nom_du_module*) peut sembler pénible à l'usage lorsque le nom du module est long. Il est alors possible de remplacer ce nom par une abréviation de son choix appelée 'alias'.

Il est recommandé d'utiliser une abréviation explicite… et d'aller voir sur le net des programmes python utilisant ce module pour garder le même alias que celui qui est généralement utilisé.

Par exemple le module numpy (bibliothèque dédiée au calcul scientifique) est souvent importée de la façon suivante :

```
>>> import numpy as np

la notation pointée se fera alors avec np au lieu de numpy :
>>> a = np.arange(15).reshape(3, 5)
>>> a
```

(… on évitera bien sûr d'appeler un module personnel 'np.py' !!!)

Devenir autonome...

Pour pouvoir travailler sur des projets, il faudra être capable de devenir autonome dans l'écriture du code. Comme on ne peut pas « s'amuser » à ré-écrire tout, on devra souvent s'appuyer sur des bibliothèques existantes, stables et vérifiées.

On peut bien sûr aller chercher des exemples de code sur le net, mais faire du copier-coller n'en assure pas la maîtrise. C'est la raison pour laquelle le programme de première vous demande d'être capable d'utiliser l'aide sur un module (help), lister son contenu (dir) et chercher de l'aide sur une fonction spécifique (help)

On pourra s'entraîner avec le module 'math', qui pourra peut-être paraître 'volumineux'…

mais pas tant que cela à côté du module pygame par exemple. Taper

```
>>> import pygame
>>> help(pygame)
```

Comme indiqué en entête de l'aide, pygame est constitué de plusieurs modules spécifiques.

Faire un dir( ) sur ce module pygame. On trouve dans la liste : 'display'.

Comment obtenir de l'aide sur 'display' :

'display' est : une constante – une fonction – un module ?

faire un dir( ) sur 'display'

On trouve dans la liste : 'set_mode'

'set_mode' est : une constante – une fonction – un module ? Comment faites-vous pour le savoir ?

Analyse d'un programme Pygame rudimentaire :

- importations :

Pourquoi a-t-on besoin de cette ligne ? (lister le contenu de pygame.locals)

```
from pygame.locals import *
```

Si l'importation avait été faite ainsi :

```
import pygame.locals
```

Quelle modification devrait-être faite dans le code ?

La ligne suivante crée un objet nommé 'screen'. Quel est son type ?

```
screen = pygame.display.set_mode((300, 50))
```

Quel est le type de l'objet 'background' créé ici :

```
background = pygame.Surface(screen.get_size())
```

Que renvoie **`screen.get_size()`** ?

Comment aurait-on pu alors écrire autrement la ligne de code ci-dessus ?

Où trouver de l'aide pour expliquer les lignes :

```
background.fill((250, 250, 250))

background.blit(text, textpos)

pygame.display.flip()
```

https://jeux.developpez.com/tutoriels/Pygame/faire-des-jeux-avec-pygame/

```
#!/usr/bin/python
# coding: utf-8

import pygame
from pygame.locals import *

def main():
    # Initialisation de la fenêtre d'affichage
    pygame.init()
    screen = pygame.display.set_mode((300, 50))
    pygame.display.set_caption('Programme Pygame de base')

    # Remplissage de l'arrière-plan
    background = pygame.Surface(screen.get_size())
    background = background.convert()
    background.fill((250, 250, 250))

    # Affichage d'un texte
    font = pygame.font.Font(None, 36)
    text = font.render("Salut tout le monde", 1, (10, 10, 10))
    textpos = text.get_rect()
    textpos.centerx = background.get_rect().centerx
    textpos.centery = background.get_rect().centery
    background.blit(text, textpos)

    # Afficher le tout dans la fenêtre
    screen.blit(background, (0, 0))
    pygame.display.flip()

    # Boucle d'évènements
    while 1:
        for event in pygame.event.get():
            if event.type == QUIT:
                return

        screen.blit(background, (0, 0))
        pygame.display.flip()

if __name__ == '__main__': main()
```

Si la dernière ligne n'était pas là, alors ce script Python ne serait constitué que

_____

Que se passerait-il alors si on tentait de le lancer ?

Quelques éléments sur la programmation événementielle

Programme de 1ère NSI : *« ..., il s'agit de montrer qu'il existe de nombreux langages de programmation, différents par leur style (impératif, fonctionnel, objet, logique, **événementiel**, etc.), ainsi que des langages formalisés de description ou de requêtes qui ne sont pas des langages de programmation. »*

Présentation

Les débuts en programmation se font souvent en utilisant la *programmation séquentielle* dans laquelle c'est le programme qui impose le fil de l'utilisation :

Programmation séquentielle :

```
annee = 2020
nom = input('Quel est ton nom ?  ')
prenom = input('Quel est ton prénom ?  ')
naissance = int(input('Tu es né(e) en quelle année ?  '))
print('Bonjour ', prenom, nom)
print(annee, " est l'année de tes : ", 2020 - naissance, " ans" )
```

```
Quel est ton nom ?  TOTO
Quel est ton prénom ?  Titi
Tu es né(e) en quelle année ?  2003
Bonjour  Titi TOTO
2020  est l'année de tes :  17  ans
```

En programmation séquentielle, il est impossible de revenir sur une valeur entrée.

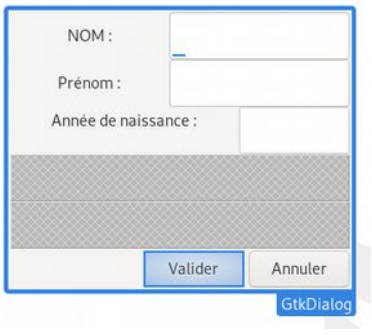
Un exemple de *programmation événementielle* avec la carte MicroBit :

```
from microbit import *

display.show('?')

while True:  # création d'une boucle infinie
    if button_a.is_pressed():
        display.show('A')
    elif button_b.is_pressed():
        display.show('B')
```

Le principe est le suivant : on fait rentrer le programme dans une boucle infinie, et à partir de là *le programme attend qu'un évènement se produise* (ici on attend que le bouton a ou le bouton b soit pressé par l'utilisateur). *Lorsqu'un événement est détecté, le programme doit réagir en conséquence.*

La programmation événementielle est particulièrement adaptée dans l'écriture de programmes pour lesquels l'interface homme machine laisse de grandes libertés à l'utilisateur :

Programmation événementielle



Ci-contre l'équivalent du premier programme, en cours de construction avec un logiciel de conception d'interface homme machine (ici le logiciel Glade).

L'utilisateur pourra entrer les données demandées dans l'ordre qui lui conviendra. Il pourra aussi revenir sur ses entrées à tout moment tant que le formulaire n'a pas été validé.

Dans ces logiciels possédant une interface graphique sur écran, les événements attendus sont, pour les plus courants, des événements souris (bouton enfoncé, relâché, mouvement) ou des événements clavier (touche enfoncée, relâchée).

Tous ces événements souris et clavier sont récupérés par le système d'exploitation qui :

- les intercepte si cela le concerne (exemple la fermeture de la fenêtre)
- ou se charge de les faire remonter vers la fenêtre active qui a son tour les intercepte ou non.

Dans l'exemple présenté, on peut au moment du clic sur le bouton « Valider » aller récupérer les données entrées dans les '*zones de texte'*, en vérifier la validité …

Ces événements s'accumulent dans une *file d'attente* ('*queue*' en anglais) qu'il faut alors traiter au niveau de l'application.

Un exemple : la gestion des évènements dans Pygame

Dans le programme donné en exemple on a :

```
    # Boucle d'évènements
    while 1:
        for event in pygame.event.get():
            if event.type == QUIT:
                return
```

Le '*package'* Pygame comprend un module dédié à la gestion des '*événements*'. C'est le module '*event*'.

Code pour afficher l'aide sur ce module : >>>

_____

Rechercher l'aide sur la fonction get( ) utilisée. Que renvoie-t-elle :

_____

Expliquer alors la signification de la ligne suivante :

_____

```
        for event in pygame.event.get():
```

peut-on changer le nom de cette variable ou doit-on laisser le mot 'event' ?

Pour mieux comprendre ce que représente la variable '*event'* définie ci-dessus, modifier la boucle infinie de la façon suivante :

```
#                   (c'est mieux que while 1 !!!)
    while True:
        for evt in pygame.event.get():
            print(evt)
```

On va regarder ce que fait cette instruction lorsque l'on bouge la souris dans (ou en dehors de) la fenêtre du programme, quand on clique sur un des boutons, quand on appuie ou que l'on relâche une touche du clavier :

<Event(5-MouseButtonDown {'pos': (252, 273), 'button': 1})> <Event(6-MouseButtonUp {'pos': (252, 273), 'button': 1})>	**appui sur le bouton de gauche de la souris**

<Event(2-KeyDown {'unicode': 'h', 'key': 104, 'mod': 4096, 'scancode': 43})> **<Event(3-KeyUp {'key': 104, 'mod': 4096, 'scancode': 43})>** **<Event(4-MouseMotion {'pos': (257, 272), 'rel': (5, -1), 'buttons': (0, 0, 0)})>** **<Event(4-MouseMotion {'pos': (271, 271), 'rel': (14, -1), 'buttons': (0, 0, 0)})>** **<Event(4-MouseMotion {'pos': (295, 268), 'rel': (24, -3), 'buttons': (0, 0, 0)})>** **<Event(4-MouseMotion {'pos': (324, 268), 'rel': (29, 0), 'buttons': (0, 0, 0)})>** **<Event(1-ActiveEvent {'gain': 0, 'state': 1})>**	**puis relâchement** **appui sur une touche ('H')** **puis relâchement** **déplacement de la souris** **la souris sort de la fenêtre**

LES EVENEMENTS : ci-dessous les événements les plus utilisés.

Valeur	Type	Attributs
1	ACTIVEEVENT	gain, state
2	KEYDOWN	key, mod, unicode, scancode
3	KEYUP	key, mod, scancode
4	MOUSEMOTION	pos, rel, buttons
5	MOUSEBUTTONDOWN	pos, button
6	MOUSEBUTTONUP	pos, button
12	QUIT	none

(http://www.pygame.org/docs/ref/event.html)

On peut récupérer le type d'un événement avec l'instruction *evt*.type que l'on peut tester avec une instruction conditionnelle :

```
      while True:
        for evt in pygame.event.get():
            if evt.type == MOUSEBUTTONDOWN: # on teste un clic de souris
                   # ...alors on décide de faire …
```

… oui mais où a eu lieu ce clic ? avec quel bouton de la souris ?

C'est là que l'on fait intervenir les attributs retournés dans une structure de données de type

_____

MouseButtonDown {'pos': (46, 15), 'button': 1}

Que fait le code suivant :

```
while True:
        for evt in pygame.event.get():
            if evt.type == MOUSEBUTTONDOWN:   # Rque : on pourrait remplacer par la
valeur 5...
                print(evt.pos)
```

Le modifier pour obtenir le type de l'attribut '*pos*' :_____ ; et celui de l'attribut '*button*' :

_____

Comment le modifier pour ne récupérer que l'abscisse du clic ? ou que l'ordonnée du clic ?

Exercice : Modifier la boucle infinie pour répondre à un évènement particulier comme par exemple :

- changer la couleur du fond lorsque l'on clique dans la zone d'affichage
- idem mais la couleur sera différente selon que l'on clique dans la moitié gauche ou la moitié droite de la zone d'affichage

- tracer une droite horizontale à l'endroit du clic
- tracer une droite verticale à l'endroit du clic
- changer la couleur du fond : en rouge si on appuie sur la touche 'R', en vert si on appuie sur la touche 'V', ou en bleu si on appuie sur la touche 'B'

Il faudra être capable de détecter un évènement de type clavier. Comment : _____

L'attribut '*key*' permettra de connaître la touche appuyée : exemple K_a pour la touche 'A'

→ voir https://www.pygame.org/docs/ref/key.html pour le détail complet !

Remarque : pour retrouver la syntaxe concernant les dictionnaires (voir le paragraphe 3.2 Opérations sur les dictionnaires dans le fascicule 'Représentation des données : Types construits') essayer ce code qui utilise la méthode '*dict*' :

```
while True:
    for evt in pygame.event.get():
        if evt.type == pygame.MOUSEBUTTONDOWN:
            print('position = ', evt.dict['pos'],
                  ' boutton = ',evt.dict['button'])
```

Quelques compléments utiles sur Pygame

Le module display

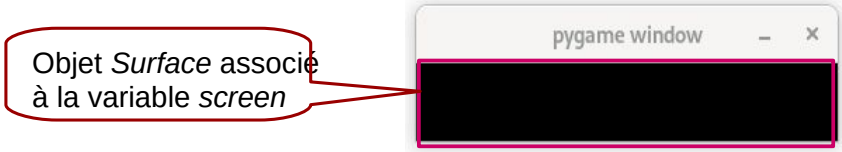
Avec Pygame, on peut créer une application en *'mode fenêtré'* ou en mode '*plein écran*'.
Ce module display sert à créer et contrôler cet espace graphique.

pygame.display.set_mode(size=(0, 0), flags=0, depth=0, display=0) → Surface

Cette fonction renvoie un objet de type *Surface* (voir paragraphe suivant). Tous les paramètres ont une valeur par défaut. On réglera ici la taille de la surface d'affichage de notre application en fournissant un tuple contenant deux entiers.

```
pygame.init()    # indispensable pour réaliser l'initialisation
screen = pygame.display.set_mode((300, 50))
```

Un objet de type Surface, ayant pour dimensions utiles 300 pixels de largeur sur 50 pixels de hauteur est créé. On sera ici en mode fenêtré. Une variable nommée 'screen' (on peut lui donner le nom que l'on veut) est associée à cet objet *Surface.*



Par défaut, l'affichage de cet objet *Surface* a un rendu de couleur noire

pygame.display.set_caption(title, icontitle=None) -> None

Cette fonction qui ne renvoie rien (*None*) permet de donner un titre à la fenêtre. Le titre est fourni sous la forme d'une chaîne de caractères :

```
pygame.display.set_caption('Programme Pygame de base')
```

`pygame.display.flip()`
Cette fonction met à jour l'intégralité de la fenêtre graphique de l'application

On peut préférer :

`pygame.display.update()`
ou qui peut mettre à jour une portion (définie par rectangle) de la fenêtre graphique :
pygame.display.update(rectangle)
ou plusieurs portions de la fenêtre graphique
update(rectangle_list) -> None

On invoquera l'une de ces fonctions de rafraîchissement d'écran *à chaque fin de tour de la boucle d'évènements*.

L'objet Surface

C'est sur un objet de type *Surface* que l'on peut dessiner, afficher des images (png, jpg, bmp, gif…), afficher du texte.

La surface de base créée au départ avec la méthode pygame.display.set_mode( ) peut servir elle aussi de support pour tout type d'affichage.

Création :

La création d'une telle surface peut se faire de façon explicite :

pygame.Surface((width, height), flags=0, depth=0, masks=None) → Surface, comme dans le programme étudié :

```
background = pygame.Surface(screen.get_size())
```

… qui créée un objet Surface de même taille que la zone graphique de l'application avec `screen.get_size()`, et lié à une variable appelée '*background*'

Remarque : cet objet va intégralement recouvrir la surface de base de l'application graphique (ici '*screen*')...qui aurait fort bien pu servir d'arrière plan en lieu et place de ce nouvel objet 'background' !

Mais un objet Surface peut également être automatiquement créé lors :

- du chargement d'une image : `pygame.image.load`('*filename*') → Surface
- de la création d'un texte : render(text, antialias, color, background=None) -> Surface

Les dimensions de cet objet Surface s'adaptent au plus juste pour contenir cette image ou ce texte.

Quelques méthodes utiles

- Remplir une surface avec une couleur :

pygame.Surface.fill(couleur) avec couleur un tuple de trois entiers compris entre 0 et 255 pour représenter dans l'ordre les couleurs rouge, vert, bleu (système RVB) :

```
background.fill((250, 250, 250))
```

- Empiler une surface sur une autre surface :

pygame.Surface.blit(source, dest, area=None, special_flags=0) → Rect

```
screen.blit(background, (0, 0))
```

Ici on vient empiler la surface ‘background’ sur la surface ‘screen’, en plaçant l’origine de background au point de coordonnées (0,0) sur la surface ‘screen’

- Récupérer des informations sur une surface :

  `get_size()` → renvoie un tuple (largeur, hauteur)

  `get_width()` ou **`get_height()`** pour ne récupérer que l’une de ses dimensions

`get_rect()` → renvoie un nouvel objet *Rectangle* ayant les dimensions de la surface sur laquelle on applique cette méthode :

Dans le code ci-dessous, la première ligne crée un objet Surface (‘text’), lors de l’utilisation de la méthode render( ) ; on crée ensuite un objet Rectangle (‘textpos’) à la ligne suivante

```
text = font.render("Salut tout le monde", 1, (10, 10, 10))
textpos = text.get_rect()
```

L’objet Rect (rectangle)

L’objet Rect permet de stoker et manipuler des aires de forme rectangulaires.

Création :

Un objet Rect peut être créé de façon explicite en fournissant les coordonnées de son point O (en haut à gauche) ainsi que sa largeur et sa hauteur :

`Rect(left, top, width, height) → Rect`
`Rect((left, top), (width, height)) → Rect` (ici on fournit deux tuples)

ou à partir d’un objet ayant les propriétés d’un objet Rect : `Rect(object) → Rect`

Utilisations

Un objet Rect possède des attributs relatifs à sa position ou ses dimensions que l’on peut modifier :

```
x,y
top, left, bottom, right
topleft, bottomleft, topright, bottomright
midtop, midleft, midbottom, midright
center, centerx, centery
size, width, height
w,h
```

Par exemple :

```
rect1.right = 10
rect2.center = (20,30)
```

ou dans le code étudié :

```
textpos = text.get_rect()
textpos.centerx = background.get_rect().centerx
textpos.centery = background.get_rect().centery
```

Il y a aussi des méthodes applicables sur un objet Rect. Ci-dessous quelques exemples extraits de la documentation :

```
move( )
     moves the rectangle
     move(x, y) -> Rect
```

Returns a new rectangle that is moved by the given offset. The x and y arguments can be any integer value, positive or negative.

```
inflate()
     grow or shrink the rectangle size
     inflate(x, y) -> Rect
```

Returns a new rectangle with the size changed by the given offset. The rectangle remains centered around its current center. Negative values will shrink the rectangle. Note, uses integers, if the offset given is too small(< 2 > -2), center will be off.

Des tests qui peuvent servir pour détecter des collisions :

`collidepoint()`
test if a point is inside a rectangle
collidepoint(x, y) -> bool
collidepoint((x,y)) -> bool

Returns true if the given point is inside the rectangle. A point along the right or bottom edge is not considered to be inside the rectangle.

Remarque : ce test peut être utilisé pour savoir savoir si la souris se trouve à l'intérieur d'un objet Rect : il suffit de passer comme paramètres à cette fonction les coordonnées de la souris. Par exemple, pour tester si la souris se trouve dans un objet Rectangle nommé zone_bt, x et y étant les coordonnées de la souris :

```
if zone_bt.collidepoint(x,y):
```

`colliderect()`
test if two rectangles overlap
colliderect(Rect) -> bool

Returns true if any portion of either rectangle overlap (except the top+bottom or left+right edges).

Afficher un texte

module : font

Pygame ne fournit pas un moyen pour écrire directement dans une surface existante : il faut créer une image du texte (de type Surface) avec la méthode Font.render() puis utiliser la fonction blit() pour l'appliquer sur une autre Surface :

`pygame.font.Font.render(text, antialias, color, background=None) → Surface`

text est une chaîne de caractère qui ne peut occuper qu'une seule ligne

color : la couleur du texte au format RVB

background : la couleur (au format RVB) d'arrière plan pour ce texte (si rien n'est défini, cet arrière plan sera transparent)

L'objet Surface renvoyé aura les dimensions nécessaires pour afficher le texte.

Le code étudié avec quelques commentaires :

```
    # création d'un objet de type 'font', on utilise la police par défaut avec une taille 36
    font = pygame.font.Font(None, 36)
    # création de l'objet Surface pour le rendu du texte
    text = font.render("Salut tout le monde", 1, (10, 10, 10))
    # placement de la Surface à l'aide d'un Rect
    textpos = text.get_rect()
    textpos.centerx = background.get_rect().centerx
    textpos.centery = background.get_rect().centery
    # la Surface 'text' est appliquée sur la Surface background avec les coordonnées fournies par textpos
    background.blit(text, textpos)
```

Afficher une image

module image

On peut charger un fichier image (type png, jpg, bmp, gif…) avec `pygame.image.load()` . Cette méthode va renvoyer un objet de type Surface.

Comme vu avant, on pourra associer un objet Rect pour manipuler les coordonnées de l'image et il faudra utiliser la méthode blit( ) pour poser cette Surface image sur une autre Surface.

Ne pas oublier de mettre à jour l'écran pour que les modifications puissent apparaître.

`pygame.image.load(filename) → Surface`

filename : le nom du fichier de type **str**

Il sera plus simple (dans un premier temps) de déposer les fichiers images nécessaires au programme dans le même dossier que celui du programme.

exemple :

```
ouahouah = pygame.image.load('chien.png')
```

crée un objet Surface ayant les dimensions nécessaires pour afficher le fichier image 'chien.png'. Cet objet Surface est associé à une variable appelée ouahouah.

```
zone_chien = ouahouah.get_rect()
```

zone_chien est alors un objet de type Rect qui permet de manipuler les coordonnées de l'image que l'on veut déposer sur la surface de travail, ce qui se fera alors avec:

```
screen.blit(ouahouah, zone_chien)
```

si l'on veut déposer l'image du chien sur la surface 'screen'

Retour sur le programme étudié

A la suite de cette étude, on peut faire un nombre de remarques non négligeables sur la façon dont ce code a été écrit. Rappelons qu'il est censé fournir au débutant un cadre de travail pour débuter avec Pygame… L'objectif est loin d'être atteint !!! Comme quoi, sur Internet, tout n'est pas bon à prendre… même dans les domaines supposés être rigoureux ! Dans ce code, cliquer dans les icônes de commentaires

```
#!/usr/bin/python
# coding: utf-8

import pygame
from pygame.locals import *

def main():
    # Initialisation de la fenêtre d'affichage
    pygame.init()
    screen = pygame.display.set_mode((300, 50))
    pygame.display.set_caption('Programme Pygame de base')

    # Remplissage de l'arrière-plan
    background = pygame.Surface(screen.get_size())
    background = background.convert()
    background.fill((250, 250, 250))

    # Affichage d'un texte
    font = pygame.font.Font(None, 36)
    text = font.render("Salut tout le monde", 1, (10, 10, 10))
    textpos = text.get_rect()
    textpos.centerx = background.get_rect().centerx
    textpos.centery = background.get_rect().centery
    background.blit(text, textpos)

    # Afficher le tout dans la fenêtre
    screen.blit(background, (0, 0))
    pygame.display.flip()

    # Boucle d'évènements
    while 1:
        for event in pygame.event.get():
            if event.type == QUIT:
                return

        screen.blit(background, (0, 0))
        pygame.display.flip()

if __name__ == '__main__': main()
```