04 | Les fonctions de PsychoPy

Sommaire

Objectifs

Vous avez appris les rudiments du langage Python dans le TP3. Vous est donc fin prêt pour entrer dans la programmation d'expériences.

Dans ce TP, vous allez découvrir les librairies, modules, et fonctions spécifiques à la programmation d'expériences en Python sous PsychoPy. Ce chapitre est particulièrement important car il contient les bases pour le prochain chapitre.

A la fin de ce chapitre :

- Vous saurez appeler et utiliser les packages essentiels pour programmer dans le Coder
- Vous connaitrez les étapes indispensables à suivre lors de l'écriture d'un script
- Vous saurez afficher des stimuli variés (texte, image, forme, son,...)
- Vous saurez attendre une réponse comportementale (touche appuyée, clics,...)
- Vous saurez mesurer un temps de réponse

Avant de commencer, assurez-vous d'avoir créé dans votre dossier coursOMN/PsychoPy/TP un sous-dossier 'TP4". Et comme d'habitude, nous vous conseillons de reproduire les différents exemples présentés dans cet eBook.

1. Librairies et modules

Tout comme R, Python utilise des *packages* ou *librairies externes*.

Comme vous l'avez vu dans le chapitre précédent, Python fait appel à des fonctions (`range()`, `del`,..). Lorsque vous les utilisez, en réalité, Python cherche, trouve et exécute des petits scripts qui codent ce que doit faire la fonction. Vous n'avez pas besoin d'ouvrir directement ces scripts, c'est Python qui gère cela.

Autant vous dire que des fonctions Python, il y a en a beaucoup ! Pour faire des stats, des maths, du traitement d'images, de sons, de mots, pour randomiser, générer des graphiques, ... Il y a une façon très codifiée de faire appel à ces fonctions, en important soit des librairies, soit des modules. Il faut comprendre la logique de base pour importer les librairies nécessaires correctement.

1.1. Importation d'une librairie

Certaines librairies sont chargées automatiquement. **D'autres doivent être importées (une seule fois) au début du script, pour pouvoir utiliser les fonctions qui y sont associées**. C'est le cas de la librairie *Pandas*, qui permet notamment d'organiser les listes python comme des dataframes de R et que nous allons utiliser pour charger des listes externes et enregistrer des données. Pour importer la librairie, il suffit de taper :

```
import pandas
```

Si l'on veut utiliser une fonction de la librairie, il faudra toujours indiquer en premier (et séparé par un point) le nom de la librairie :

```
pandas.read_table("listeExterne.txt")
```

Ici, la fonction cible est `read_table()`. Elle est utilisable si elle est précédée du nom de la librairie. Comme il peut être un peu lourd de devoir écrire toujours le nom de la librairie, on peut lui donner un alias au moment de l'importation et ensuite n'utiliser que cet alias :

```
import pandas as pd
pd.read_table("listeExterne.txt")
```

1.2. La librairie PsychoPy

'PsychoPy' n'est en fait rien d'autre qu'une librairie Python parmi d'autres (et qui en principe peut être utilisée dans d'autres interfaces que l'application PsychoPy). C'est une librairie dédiée à la programmation d'expériences pour la psychologie, les neurosciences et les sciences cognitives. Nous n'avons pas besoin de l'importer à chaque fois car c'est automatiquement fait en lançant le logiciel PsychoPy.

Par contre, il y a des modules de PsychoPy qu'il faudra importer et qui sont critiques pour programmer une expérience. Il en existe beaucoup, et voici les principaux que nous allons utiliser:

- **visual**: Contient des fonctions pour créer la fenêtre virtuelle (cf. plus bas) et tous les stimuli visuels !
- **core**: Contient des fonctions de base pour faire tourner une expérience (timing et fermeture du programme)
- **event**: Contient des fonctions pour utiliser le clavier et la souris
- **gui**: Contient des fonctions pour enregistrer des metadata facilement
- **sound**: Contient des fonctions pour jouer du son

Gardez en mémoire qu'il y a beaucoup plus de modules que ça. Nous n'allons voir que les principaux, mais vous pourriez être intéressé par d'autres fonctions, d'autres modules ('voicekey' pour le traitement en temps réel de stimuli audio captés, 'parallel' pour interagir avec le port parallèle si vous faites de l'EEG par exemple, 'microphone' pour capter et analyser du son...). Dans ce cas, votre livre de chevet doit être le manuel de PsychoPy qui contient tout le descriptif des fonctions, organisées par modules, avec la liste précise des arguments.

La ligne de commande à insérer en début de script pour importer ces modules sera structurée ainsi :

```
# importation d'un module :
from psychopy import core

# importation de plusieurs modules :
from psychopy import core, data, gui
```

Exercez-vous !

2. Logique de programmation

Programmer dans le Coder nécessite de réaliser un certain nombre d'étapes indispensables. Nous allons voir les étapes principales ici, et d'autres s'ajouteront au fur et à mesure que nous avancerons dans ce chapitre et le suivant.

Attention, pour bien programmer, il vous suffit non seulement d'inclure ces étapes mais surtout de mettre en relief cette ossature pour que votre script soit facilement lisible. C'est pourquoi nous prendrons toujours l'habitude de mettre des commentaires (lignes commençant par '#') pour marquer les différentes parties du script. Nous vous conseillons d'en faire autant !

2.1. Etape 1: Initialiser

L'initialisation comprend des commandes pour préparer l'écriture du script, telles que :

- l'encodage des informations (UTF-8) pour gérer les marques diacritiques (cela permettra notamment d'écrire des commentaires avec des accents)
- l'importation de modules

Voici un exemple:

```
# -*- coding: utf-8 -*-

# Initialisation
from psychopy import visual, core
```

Pourquoi s'embêter avec UTF-8 ?!

Vous allez voir qu'à plusieurs endroits nous allons revenir à l'encodage UTF8. Comme déjà indiqué, les applications développées en anglais rencontrent souvent des problèmes quand du texte avec des marques diacritiques sont insérées.

Pour dépasser cela, on inclut généralement une demande pour que le texte soit encodé au format UTF8.

L'UTF8 (abréviation de l'anglais *Universal Character Set Transformation Format - 8 bits*) est un codage de caractères informatiques conçu pour coder l'ensemble des caractères du « répertoire universel de caractères codés », autrement dit tout caractère existant. Comme la langue française est remplie de caractères avec des marques diacritiques, il faut toujours se préoccuper d'insérer des lignes de commande spécifiant un encodage UTF8.

2.2. Etape 2: Ouvrir la fenêtre virtuelle

Le principe de programmation dans PsychoPy consiste à **ouvrir une fenêtre virtuelle dans laquelle vous 'dessiner' des stimuli**. Quand tout est prêt, cette fenêtre est affichée sur l'écran.

La règle est donc simple: à chaque fois que vous voulez faire apparaître un stimulus, il faut le créer dans la fenêtre virtuelle puis afficher cette fenêtre. L'expérience commence donc par la création de cette fenêtre. Il suffit d'utiliser la fonction `Window()` dans le module `visual`. Vous pouvez indiquer un ensemble de paramètres : sa taille, la couleur, l'unité de mesure,... La fonction `visual.Windows()` est la fonction qui sert à définit la fenêtre, la fenêtre devant être assignée à une variable (par ex. `w`). C'est cette variable qui représente directement la fenêtre virtuelle et nous allons toujours y faire référence (bien sûr, vous pouvez utiliser un autre nom de variable !).

```
# Ouverture fenêtre virtuelle
w = visual.Window([800,600],color="black",units='pix')
```

Notez qu'ici la fenêtre aura une taille de 800x600 pixels. Si vous voulez qu'elle prenne tout l'écran (ce qui est le cas lors du vrai testing), alors il faudrait entrer la résolution de votre écran. En fait, il vaut mieux utiliser l'argument `fullscr=True`.

```
# Ouverture fenêtre virtuelle
w = visual.Window(fullscr=True,color="black",units='pix')
```

Rappelez-vous : quand vous êtes en phase d'écriture du script, c'est plutôt bien d'utiliser une fenêtre réduite (vous pouvez plus facilement reprendre la main sur votre ordinateur en cas de plantage). Mais **lorsque vous êtes en phase de test pilote et de testing réel, il est indispensable d'utiliser la fenêtre entière !**.

2.3. Etape 3: Créer un stimulus dans la fenêtre

Nous verrons dans la section suivante le détail des commandes pour créer tout type de stimulus. Ici nous allons prendre un exemple. Disons que nous voulons afficher un carré rouge. Voici la commande :

```
# Stimulus
stim = visual.Rect(w, size=[200,200], fillColor="red", lineColor="red")
```

Ici, nous définissons un rectange (`visual.Rect()`) dans la fenêtre `w` que nous nous assignons à la variable `stim`. Ce rectange est de taille 200x200 pixels (`size=[200,200]`), coloré en rouge (`fillColor="red"`), avec un contour rouge (`lineColor="red"`).

Une fois que l'objet est créé, il faut le 'dessiner' dans la fenêtre. Pour cela on utilise la fonction `draw()` que l'on applique à l'objet que l'on a créé (`stim`) :

```
# Ouverture fenêtre virtuelle
stim.draw()
```

2.4. Etape 4: Afficher la fenêtre

Une fois que vous avez créé la fenêtre et dessiné dedans tous les objets que vous voulez afficher, vous devez afficher la fenêtre. Pour cela, il faut utiliser la fonction `flip()` que l'on applique à notre objet fenêtre (`w` ici) :

```
w.flip()
```

Admettons ensuite que vous voulez que ce stimulus reste à l'écran pendant 1.5 secs, c'est la fonction `wait()` du module `core` qui vous permet de faire cela:

```
core.wait(1.5)
```

Attention ! Il est très important ici de comprendre que **tous les objets** qui ont 'subi' la commande `draw()` avant la commande `flip()` vont apparaître sur l'écran. Dès que la commande `flip()` a fini d'être exécutée, ces objets sont affichés sur l'écran mais **en arrière plan, la fenêtre virtuelle est vidée**. Autrement dit, `flip()` affiche ce qui a été dessiné dans la fenêtre mais la fenêtre 'oublie' aussitôt ce qui avait été dessiné. Prenons un exemple:

```
01    stim1.draw()
02    stim2.draw()
03    w.flip()
04    stim3.draw()
05    w.flip()
```

Lorsque la commande `flip()` de la ligne 3 est exécutée, il y a deux objets présentés dans la fenêtre (`stim1` et `stim2`), car tous les deux ont été dessinés avant l'éxécution de `flip()`.

Lorsque la commande `flip()` de la ligne 5 est exécutée par contre, comme il n'y a eu qu'un objet dessiné dans la fenêtre depuis le dernier flip (`stim3`) seul `stim3` sera affiché. **L'erreur serait de croire que PsychoPy garde en mémoire les objets précédents et que l'exécution de la ligne 5 produit l'affichage de stim1, stim2, et stim3**. Ce n'est pas du tout le cas. Au contraire, retenez que quand `flip()` a été exécuté, la fenêtre virtuelle est de nouveau vide à la fin de l'exécution.

L'ordre compte !

Dans le Builder, nous avons vu que l'agencement des objets dans une routine peut détermer l'ordre d'affichage des objets. C'est la même chose avec le Coder. Admettons que vous voulez afficher un carré rouge de 200 pixels de côté au centre de l'écran et que vous voulez mettre par dessus un carré jaune de 100 pixels, afin de créer un stimulus comme sur la figure en bas.

Le point important à comprendre est que lorsque vous dessinez plusieurs objets dans la fenêtre, PsychoPy les dessine séquentiellement. Donc, si vous utilisez la commande draw() d'abord pour le carré rouge, puis pour le carré jaune, le carré jaune va apparaître après le carré rouge, donc SUR lui :

carreRouge.draw()

carreJaune.draw()

w.flip()

Si vous utilisez la commande draw() d'abord pour le carré jaune, il sera au final recouvert par le carré rouge. Or comme le carré rouge est plus grand que le carré jaune, le carré jaune n'apparaitra pas visuellement, puisqu'il sera caché par le carré rouge

carreJaune.draw()

carreRouge.draw()

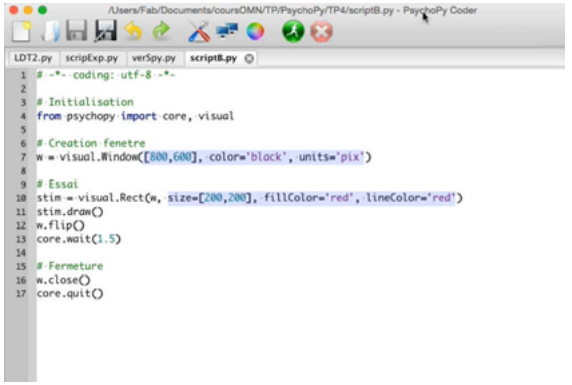
w.flip()

2.5. Etape 5: Fermer la fenêtre

Une fois que votre expérience est terminée, il faut fermer la fenêtre virtuelle et nettoyer :

```
# Fermeture
w.close()
core.quit()
```

Vous connaissez à présent les étapes clé pour écrire un script sous le Coder ! Regardez la *Vidéo 1* pour quelques conseils supplémentaires.



Vidéo 1.– Récapitulatif des étapes d'écriture d'un script

Erreurs fréquentes

Voici une liste des erreurs fréquentes à ce stade, avec le message d'erreur qui apparait dans la fenêtre output :

1) Oublier de spécifier la fenêtre lors de l'utilisation d'une fonction pour créer un objet: message: le script plante avec comme message d'erreur *TypeError: __init__() takes at least 2 arguments (1 given)*

KO: vous avez écrit stim = visual.Rect(size=[200,200])
OK: au lieu de stim = visual.Rect(w,size=[200,200])

2) Oublier les parenthèses à la fin d'une fonction message: le script s'arrête sans message d'erreur

KO: vous avez écrit w.flip ou stim.draw
OK: au lieu de w.flip() ou stim.draw()

3) Mal écrire le nom d'un argument d'une fonction (ici fillColor) message: le script plante avec comme message d'erreur TypeError: __init__() got an unexpected keyword argument 'colorFill'

KO: vous avez écrit stim = visual.Rect(fillColor="red") ou (oubli d'une majuscule)
OK: au lieu de stim = visual.Rect(colorFill="red")

4) Mal écrire le nom d'une fonction (ici visual.Rect()) message: le script plante avec comme message d'erreur AttributeError: 'module' object has no attribute 'rect'

KO: vous avez écrit stim = visual.rect(win,size=[200,200]) ou (nom erroné)
OK: au lieu de stim = visual.Rect(win,size=[200,200])

Exercez-vous !

3. Créer des objets

Nous allons voir dans cette section comment créer des stimuli. Seuls les plus courants vont êtes vus. Là, encore si vous avez besoin de générer des stimuli plus sophistiqués, le manuel de PychoPy sera votre ami. De même, les fonctions que nous allons utiliser peuvent inclure beaucoup d'arguments.

Tous les paramètres que vous pouviez entrer dans le Builder (par exemple l'orientation d'une forme géométrique, son nombre de vertex, son opacité, sa couleur, sa position,...) sont des arguments potentiels des différentes fonctions que nous allons voir. Là encore, nous n'allons pas les détailler, mais le manuel de PychoPy doit être votre référence. Nous le mettons sur l'UV pour qu'il vous soit facilement accessible.

Enfin, dans les exemples ci-après notez que les commandes `[objet].draw()` et `[fenêtre].flip()` ne sont pas incluses par souci de clarté, et que la variable correspond à la fenêtre virtuelle est `w`.

3.1. Texte

La fonction à utiliser est `TextStim()`. En règle général, utiliser cette commande met un peu plus de temps pour l'ordinateur que les fonctions pour les autres stimuli visuels. Ce sont notamment les arguments (ou attributs) qui affectent la forme des lettres qui prennent un peu plus de temps (*text*, *height*, *font*, *bold*...).

```
# Ex1
stim = visual.TextStim(w, text='Hello!')
```

```
# Ex2
stim = visual.TextStim(w, text=u"vélo")

# Ex3
stim = visual.TextStim(w, text='Hello!',pos=[-200,12],
height=30)
```

Remarquez que si vous voulez afficher un mot isolé qui contient un accent, il faut le faire précéder de **u** (pour spécifier qu'il est au format utf-8 encore une fois !).

3.2. Forme géométrique

Voici plusieurs exemples pour créer une forme géométrique, chacune utilisant une fonction différente. Pour plus de paramètres (ou pour comprendre ceux présentés ici, voir le manuel de PsychoPy en ligne ou sur l'UV. Pour explorer ces fonctions, vous pouvez créer un script avec les commandes présentées dans la section *Section 2 page 7* et créer/afficher les stimuli suivants :

3.2.1. Carré/rectangle

```
stim = visual.Rect(w, size=[200,200], fillColor="red", line-
Color="red")
```

3.2.2. Cercle

```
stim = visual.Circle(w, radius = 120, edges=75, fillCo-
lor="blue", pos=[-150,12])
```

3.2.3. Ligne

```
stim = visual.Line(w, start=[-200,12],end=[21,78])
```

3.2.4. Polygone

```
stim = visual.Polygon(w, radius = 120, edges=3, fillCo-
lor="blue", pos=[-150,12])
```

3.2.5. Forme non régulière

```
stim = visual.ShapeStim(w, lineWidth = 3, vertic
es=[[-45,-78],[-78,212],[99,24],[3,7]])
```

3.3. Images

La fonction à utiliser est **ImageStim()**. Voici quelques exemples (là encore, vous pouvez faire des essais avec le fichier img1.jpg disponible sur l'UV):

```
# Ex1
stim = visual.ImageStim(w, image = 'img1.jpg')

# Ex2
stim = visual.ImageStim(w, image = 'img1.jpg',co-
lor=[1,0.5,0.5])

# Ex3
stim = visual.ImageStim(w, image = 'img1.jpg', opacity=0.5)
```

3.4. Vidéo

La fonction à utiliser est `MovieStim()` :

```
stim = visual.MovieStim3(w, filename = 'video.mp4')

while stim.status != visual.FINISHED:
  mov.draw()
  w.flip()
```

Plusieurs choses à noter ici :

- Il ne faut pas oublier le 3 à la fin de `MovieStim` (le concepteur a fait plusieurs versions de cette fonction, et c'est la dernière qui est censée bien fonctionner).
- Contrairement aux autres stimuli vus ici, une vidéo est en fait une suite d'images, qu'il va donc falloir afficher les unes après la autres. Pour cela, nous allons utiliser une boucle WHILE.
- La définition de la boucle (`stim.status != visual.FINISHED`) est : *tant que le statut de la vidéo n'est pas égal à FINI* (autrement dit, *tant que la vidéo n'est pas finie*), alors on affiche les images qui la composent.

3.5. Son

L'ensemble des fonctions utilisées précédemment étaient dans le module `visual` (car ce sont tous des objets visuels). Les fonctions pour jouer du son sont dans un autre module :

```
from psychopy import sound

s = sound.Sound(value="beep1.wav")
s.play()
core.wait(2.0)
```

Vous noterez qu'ici on n'utilise pas `draw()` et `flip()`.

Exercez-vous !

4. Mettre fin à un événement

Vous savez maintenant afficher des stimuli. C'est un bon début. Généralement, vous voulez aussi définir la fin d'affichage/d'exécution d'un stimuli. Là encore, nous allons voir uniquement les trois manières standard de faire (fin par le temps, fin par un clic de souris, fin par un appui sur une touche de clavier). Pour cela, nous allons utiliser le module `event` de PsychoPy.

Attention, vous pourriez utiliser quelque chose d'autre : une clé vocale ou une button box (bien plus recommandée qu'un clavier !). Là encore, consultez le manuel de PsychoPy.

4.1. Temps

4.1.1. core.wait()

Nous avons déjà vu la fonction `wait()` du module `core`. Admettons que vous voulez qu'une image n'apparaisse que 100 msecs. Après son affichage (`w.flip()`), introduisez une pause de 0.1 sec :

```
core.wait(0.1)
```

Ici, le stimulus resterait affiché 0.1 sec. En réalité, **cette fonction peut se révéler très imprécise**. Il n'est donc **pas recommandé de l'utiliser pour de réelles expériences** (mais elle fait l'affaire quand vous vous entraînez).

4.1.2. Horloge et boucle while()

Une façon propre d'afficher un stimulus un certain temps est de demander à PsychoPy de l'**afficher tant qu'un certain temps ne s'est pas écoulé**. Pour cela, on fait appel à l'horloge interne de l'ordinateur, supposée être précise.

Vous avez vu que j'ai utilisé l'expression *tant que*. Nous allons donc utiliser une boucle WHILE:

```
01    horloge = core.Clock()
02    while horloge.getTime() < 0.1:
03      o.draw()
04      w.flip()
```

Détaillons ces lignes de commandes:

- `core.Clock()` sert à obtenir l'heure (via l'horloge interne de l'ordinateur). La ligne 1 donne donc un temps de départ, enregistré dans une variable nommée `horloge` (et qui pourrait donc porter n'importe quel nom).

- `getTime()` sert de nouveau à obtenir l'heure de l'ordinateur. Si on fait la soustraction entre la deuxième heure et la première heure, on devrait obtenir le temps écoulé entre les deux. En fait, en appliquant `getTime()` à l'objet `horloge`, on obtient directement le résultat de cette soustraction et donc une mesure du temps qui s'est écoulé entre la ligne 2 et la ligne 1.

- La boucle sert à dire: *tant que le temps écoulé est inférieur à 100 ms*, alors dessine l'objet (*o*) dans la fenêtre (*w*) et affiche-la. Dès que le temps sera supérieur à 0.1 sec, alors la boucle s'arrêtera et le script exécutera les actions suivantes. Si la fenêtre est ensuite vidée, alors l'objet sera bien resté juste 100 msecs à l'écran.

Cette méthode est plus précise que `core.wait()`, c'est donc elle que nous allons privilégier. Sachez néamoins que la précision de la boucle WHILE est loin d'être parfaite. La meilleure méthode à suivre est de parler en *frames*. Le chapitre 2 avait présenté ce concept et montré son application dans un script Builder (section optionnelle). Le présent chapitre présente également (de manière optionnelle) comment utiliser les frames dans le Coder. **J'insiste sur le fait que c'est LA bonne méthode pour gérer correctement le timing**, tout particulièremenent si vous faites des présentations rapides de stimuli.

Horloge interne ?!

L'horloge interne est un processus qui permet de donner l'heure et la date aux programmes qui le demandent. Comment est-ce possible qu'un ordinateur débranché et non relié à un réseau quelconque puisse se souvenir de l'heure et de la date ? Dans la carte mère de l'ordinateur, il existe une pile fournissant l'énergie nécessaire au fonctionnement de l'horloge interne voir ici.

4.2. Appui sur le clavier

4.2.1. Remettre à zéro

Lorsque vous attendez une réponse au clavier, il est toujours bon –avant d'afficher le stimulus– de nettoyer le buffer qui stocke les touches appuyées ou clics de la souris des précédents essais. La commande est la suivante (elle nécessite d'avoir charger le module `event` de PsychoPy) :

```
event.clearEvents()
```

4.2.2. waitKeys()

Cette fonction permet d'attendre un appui sur une touche, mais elle stoppe tout (y compris le dessin des objets dans la fenêtre virtuelle) en attendant une réponse. Vous affichez par exemple un objet (`o`) qui reste affiché jusqu'à ce que le participant appuie sur une touche :

```
event.clearEvents()

o.draw()
w.flip()
event.waitKeys()
```

Dans ce cas, il ne se passera rien tant que le participant n'aura pas appuyé sur une touche, quelle qu'elle soit.

Vous pourriez préciser que vous attendez un appui sur une touche déterminée (ici, Y ou O):

```
event.clearEvents()

o.draw()
w.flip()
event.waitKeys(keyList=['y','o'])
```

Attention: **PsychoPy raisonne ici avec un clavier Querty et non Azerty !** Référez-vous donc à l'annexe 7 pour connaître la correspondance des touches !

Vous pouvez aussi dire que vous êtes enclin à attendre une réponse seulement un certain temps:

```
event.clearEvents()

o.draw()
w.flip()
event.waitKeys(maxWait=2, keyList=['y','o'])
```

4.2.3. getKeys()

`getKeys`est une fonction très similaire à `waitKeys()`, la différence étant que tout ne s'arrête pas quand l'ordinateur rencontre `getKeys`. Lorsque cette commande est rencontrée, l'ordinateur balaie simplement l'ensemble des touches du clavier pour voir si l'une d'entre elle a été pressée, et une fois fait, il continue de lire les lignes de code suivantes.

Ainsi, pour qu'elle ait un intérêt, cette fonction doit nécessairement être utilisée dans une boucle (WHILE, FOR) puisqu'elle ne fait que vérifier 1 fois quelle touche a été pressée. Si on met

cette commande dans une boucle WHILE, alors la vérification se fera jusqu'à une certaine condition. Voici un exemple :

```
event.clearEvents()
while True:
  o.draw()
  w.flip()
  if event.getKeys('y'):
    break
```

`while True` permet de dire de **fait tourner la boucle indéfiniment** (on aurait pu écrire aussi `while 1`). A chaque passage, l'objet est créé, dessiné et affiché. Une fois cela fait, PsychoPy vérifie si pendant ce temps la touche Y a été pressée (test IF). Si c'est le cas, le mot outil `break` permet de sortir de la boucle WHILE (sinon on reste bloqué dedans indéfiniment !).

Dans ce cas, nous aurions tout à fait pu utiliser des lignes de code avec `waitKeys()` (cf. section plus haut), mais nous verrons que `getKeys()` se révèlent parfois indispensable.

4.2.4. Récupérer les clés de réponse

Très souvent, nous voulons savoir sur quelle touche le participant a appuyé. C'est tout à fait possible puisque les fonctions `waitKeys()` et `getKeys()` ont été construites de façon à enregistrer l'identité des touches appuyées.

Voici une façon de faire avec `wait.Keys()` :

```
01    event.clearEvents()
02    o.draw()
03    w.flip()
04    rep = event.waitKeys(keyList=['y','o'])
05    print rep
```

A la ligne 5, la variable `rep` est créée pour stocker la touche de réponse appuyée par le participant. Après l'essai, voici un contenu possible de `rep`:

```
print rep
> ['o']
```

Admettons que durant un temps limité (disons 5 secs), le participant peut appuyer sur n'importe quelle touche. Une fois le temps écoulé, vous voulez connaitre l'identité des touches appuyées. Voici le code pour faire cela :

```
01    allRep = []
02    event.clearEvents()
03
04    H = core.Clock()
05    while H.getTime() < 5:
06      o.draw()
07      w.flip()
08      rep = event.getKeys()
09      allRep = allRep + rep
10
11    print allRep
```

Décodons ces lignes.

- On crée tout d'abord un vecteur vide (`allResp`) dans lequel l'identité des touches pressées va s'incrémenter. Il doit nécessairement être vide au début puisque aucune touche n'a encore été pressée.
- `H` est créé pour coder l'heure de l'ordinateur. Sur base de cela, on peut créer une boucle WHILE de sorte à ce que ce qui sera mis dedans s'exécutera pendant un certain temps (cf section *Section 4.1.2 page 17*).
- A chaque passage dans la boucle, s'il y a eu une ou des touches pressée, elles sont stockées dans `rep` (ligne 8), dont le contenu est lui-même ajouté au vecteur `allRep` (ligne 9). La différence entre `rep` et `allRep` est que `rep` stocke la ou les touches pressées lors d'un passage dans la boucle alors que `allRep` stocke les touches pressées pendant l'entièreté de la durée de la boucle (5 secs ici).

Voici ce que j'ai obtenu dans l'Output:

```
> ['d', 'k', 'j', 's', 'j', 'f', 'k', 'd', 'j', 'f', 'i', 'e',
'o', 'u', 'i', 'j', 'bracketleft']
```

Vous pouvez donc en déduire que j'ai appuyé frénétiquement sur mon clavier pendant les 5 secs d'apparition du stimulus.

4.3. Clic de souris

4.3.1. Fonctions générales

L'utilisation de la souris commence par la création d'un objet 'souris' (via la fonction `event.Mouse()`) sur lequel on va pouvoir appliquer un certain nombre de fonctions :

```
maSouris = event.Mouse()
```

A tout moment, vous pouvez rendre la souris visible ou non à l'écran :

```
maSouris.setVisible(1) # visible
maSouris.setVisible(0) # invisible
```

Vous pouvez également positionner la souris quelque part sur l'écran (ici au centre de l'écran) :

```
maSouris.setPos([0,0])
```

4.3.2. Savoir s'il y a eu des clics

La fonction `getPressed()` appliquée à l'objet souris permet de retourner l'état des trois boutons de la souris sous forme d'un vecteur (avec 0 = bouton non pressé, 1 = bouton enfoncé). Le vecteur `[0,0,0]` indique ainsi qu'aucun des trois boutons de la souris (respectivement : clic gauche, molette, clic droit) n'a été pressé. Le vecteur `[1,0,0]` indique que le bouton gauche a été pressé, etc...

Dans l'exemple suivant, la réponse sera **[0,0,0]** si l'on tape cette commande, car au moment de la lecture (qui est très court) aucun bouton n'a été pressé :

```
01    o.draw()
02    w.flip()
03    print maSouris.getPressed()
```

Dans cet exemple, c'est en réalité quasi-impossible (ou du moins très aléatoire) d'espérer que le participant clique exactement au moment de la ligne 3.

Ainsi, **tout comme `getKeys()`, `getPressed()` doit s'utiliser dans une boucle WHILE**. Ainsi, effectivement si on laisse un peu plus de temps (ici 1 sec), les choses changent :

```
01    H = core.Clock()
02    while H.getTime() < 1:
03      o.draw()
04      w.flip()
05      print maSouris.getPressed()
```

L'output est très long (~ 60 lignes). Chaque ligne correspond à une période de 17 msecs environ (c'est-à-dire à un *frame*. Si vous souhaitez comprendre pourquoi, lisez la partie *Gestion avancées du timing* dans l'eBook 2) :

```
...
[0, 0, 0]
[1, 0, 0]
[1, 0, 0]
[1, 0, 0]
[1, 0, 0]
[1, 0, 0]
[1, 0, 0]
[0, 0, 0]
[0, 0, 0]
...
```

Ce que l'on peut constater, c'est que –alors que je n'ai cliqué qu'une fois–, le clic gauche est marqué sur 5 périodes. Cela est dû au fait que ça prend du temps de faire un clic (plus que 17 ms !). Ici, cela m'a pris environ 85 msecs (17 x 5) pour enfoncer une touche de souris et la relâcher.

4.3.3. Attendre un clic de souris

Très souvent, nous voulons que l'affichage de l'écran disparaisse une fois que le participant a cliqué. Pour faire cela, il suffit d'adapter les lignes précédentes :

```
01    # Limite de temps infinie pour répondre
02    while True:
03      o.draw()
04      w.flip()
05      if maSouris.getPressed()[0]==1:
06        break
```

Décomposons ces lignes:

- On utilise une boucle WHILE pour que la commande **`getPressed()`** soit utilisable. Ces lignes disent 'la boucle tourne indéfiniment (`**`while True`**), mais si un clic gauche est détecté (**`if maSouris.getPressed()[0]==1:`**), alors l'ordinateur quitte la bouche (**`break`**)'.

Comme nous l'avons vu précédemment, `maSouris.getPressed()` retourne un vecteur. L'utilisation de `[0]` permet d'indexer la première position du vecteur (qui correspond au clic gauche). Si le premier élément du vecteur vaut 1, alors cela signifie qu'il y a eu un clic gauche, et donc on peut arrêter la boucle puisque le participant a cliqué.

Voici un second exemple avec une limite de temps :

```
# Limite de 2 secs pour répondre
H = core.Clock()
while H.getTime() < 2:
  o.draw()
  w.flip()
  if maSouris.getPressed()[0] == 1:
    break
```

4.3.4. Connaitre la position des clics

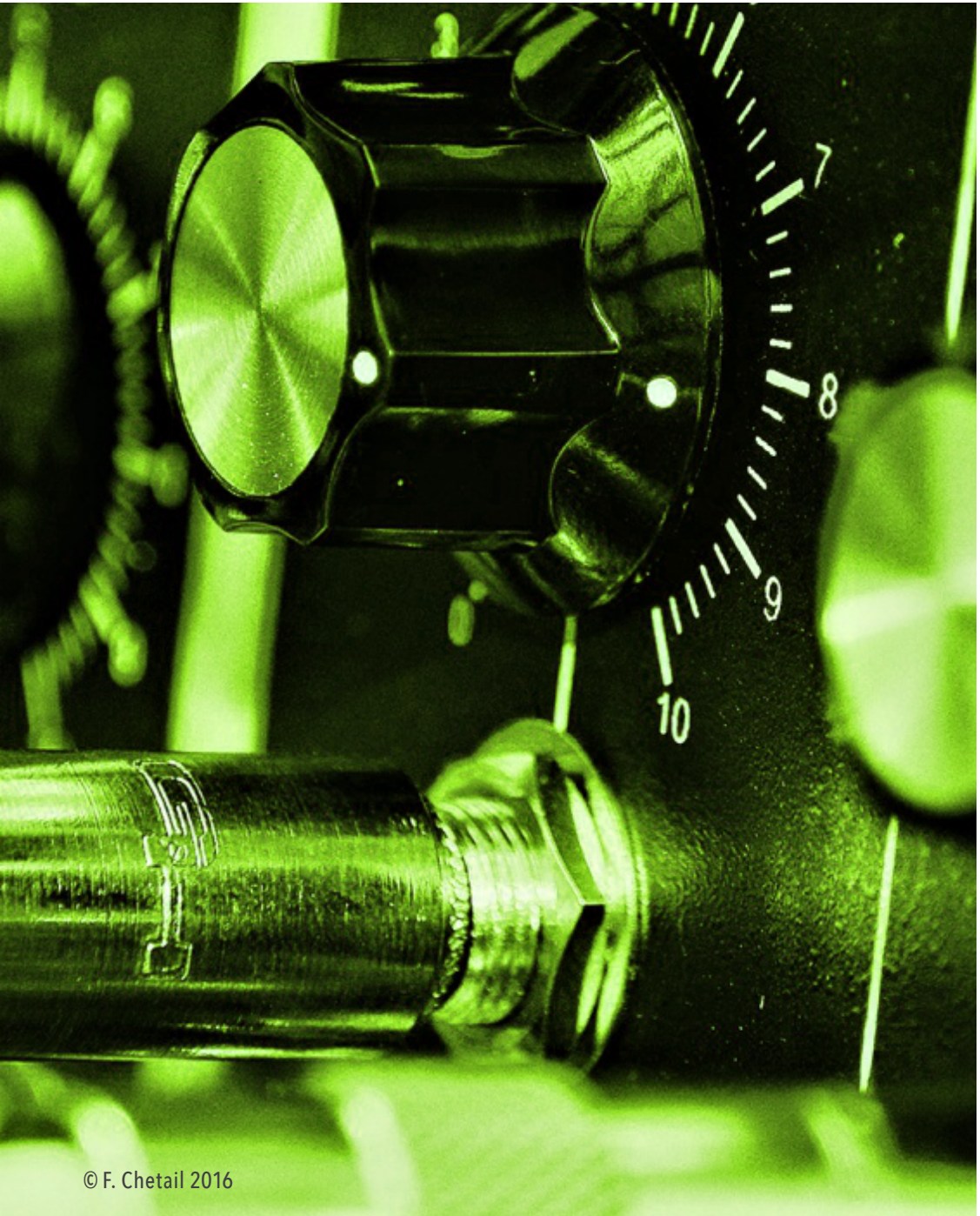
Grâce à la fonction `getPos()`, vous pouvez extraire aussi la position du clic:

```
while True:
  o.draw()
  w.flip()
  if maSouris.getPressed()[0]==1:
    print souris.getPos()
    break
```

La résultat est un vecteur contenant les coordonnées (x,y) de la position du clic :

```
[-43 88]
```

5. Enregistrer des mesures



Nous avons vu les commandes de base pour enregistrer les touches de réponse (clic et bouton pressés sur le clavier). Evidemment, ce qui va nous intéresser surtout c'est le temps mis pour répondre.

Voici la logique de la procédure à utiliser pour enregistrer les temps de réponse, avec laquelle vous êtes déjà familiers :

- on demande l'heure de l'ordinateur au moment où le stimuli est affiché (temps start) en utilisant la fonction `Clock()`.
- on demande l'heure de l'ordinateur au moment où le participant donne une réponse (temp stop). Pour cela on utilise `getTime()` qui calcule automatiquement le temps écoulé entre start et stop.

Voici un exemple avec le clavier. L'ensemble des commandes doit vous être familier. La seule différence avec ce que vous avez vu avant est qu'on stocke le résultat de `get.Time()` dans une variable (`rt`). Ce temps correspond ici au temps de réponse (le temps écoulé entre l'apparition du stimuli –ligne 6– et l'appui sur la touche Y –ligne 7)

```
01    event.clearEvents()
02
03    H = core.Clock()
04    while True:
05      o.draw()
06      w.flip()
07      if event.getKeys('y'):
08        rt = H.getTime()
09        break
```

Attention, le temps donné par `getTime()` est **en secondes**, donc si vous voulez voir le résultat en millisecondes, il faut multiplier par 1000 :

```
print rt*1000


> 1024.58475
```

Voici un autre exemple avec la fonction `waitKeys()` cette fois :

```
01    event.clearEvents()
02
03    o.draw()
04    w.flip()
05    H = core.Clock()
06    event.waitKeys()
07    rt = H.getTime()
```

Enfin, voici un exemple avec une souris, qui suit les mêmes principes :

```
01    souris = event.Mouse()
02
03    H = core.Clock()
04    while True:
05      o.draw()
06      w.flip()
07      if souris.getPressed()[0]==1:
08          rt = H.getTime()
09        break
10    print rt*1000
```

```
11
12    > 1024.58475
```

Attention, **lorsqu'une boucle while est utilisée, H doit être créé AVANT la boucle, et non à l'intérieur avant `w.flip()`, car sinon H va être ré-actualisé à chaque passage dans la bouche et donc on aura perdu la vraie heure de début de l'apparition du stimulus.**

Par contre, sans boucle (cf. exemple avec `waitKeys()`), `H` doit toujours être créé après l'affichage de la fenêtre.

Exercez-vous !

6. Prêt(e) pour le TP ?

Ma check-list

- ☐ Je sais expliquer le lien entre librairie, module et fonction.
- ☐ Je sais énoncer les grandes étapes à inclure pour écrire une expérience dans le Coder.
- ☐ Je sais créer des objets variés (texte, images, formes,...).
- ☐ Je sais utiliser le manuel PsychoPy pour trouver la syntaxe à utiliser pour modifier les paramètres par défaut de mon objet créé (changer l'orientation du texte par exemple).
- ☐ Je sais utiliser les boucles FOR, WHILE et l'outil IF pour terminer la présentation d'un objet (par le temps, le clavier ou la souris).
- ☐ Je sais récupérer le temps de réponse.

7. L'organisation des modules/packages (Optionnel)

Comme dit précédemment, il y a une façon très codifiée de faire appel à des fonctions Python, en important soit des librairies, soit des modules.

7.1. Importation

Réfléchissez : généralement, vous ne rangez pas tous vos fichiers au même endroit sur votre ordinateur. Les fichiers similaires sont rangés dans un dossier (par exemple, vous mettez tous vos fichiers sons dans un dossier Musique, tous vos fichiers photos dans un dossier Images, etc...).

C'est la même logique pour Python : il y a des librairies (*packages*) pour les dossiers et des *modules* pour les fichiers. De la même façon que vous pouvez avoir dans votre hiérarchie des sous-dossiers (sous dossiers Jazz, Rock, Ska dans le dossier Musique), Python peut avoir des *sub-packages*.

Un répertoire doit contenir un fichier nommé **init**.py (qui peut être vide) pour que Python le considère comme un package ou comme un sous-package. La *Figure 1* illustre cela. La librairie Game contient 3 sous-packages (Sound, Image, Level), chacun contenant 3 modules. Ces modules contiennent des fonctions que l'on peut utiliser dans un script.

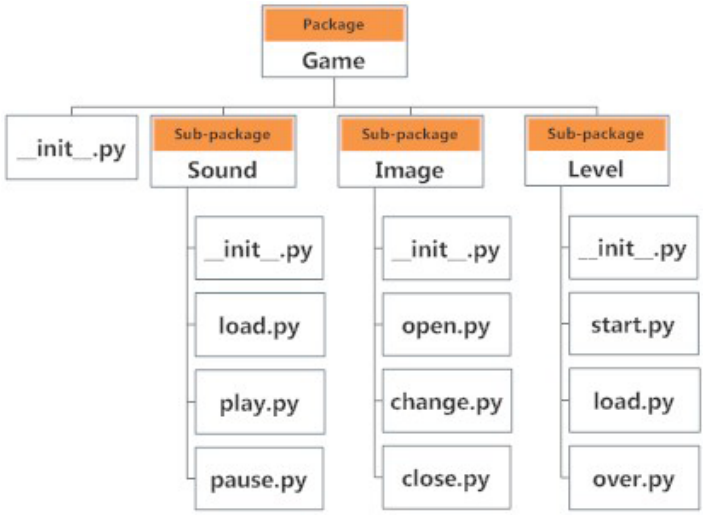


Figure 1.– Packages et modules en Python

Vous avez utilisé quelques fonctions Python dans le chapitre 3 sans importer de packages ou de modules. Néanmoins, la plupart du temps cela va être nécessaire.

7.1.1. Importation de tout un package

Pour importer, on va toujours utiliser le mot clé `import`. Attention, ici comme après, il faut toujours respecter la casse des noms des packages et modules ! (n'oubliez pas les majuscules donc). Prenons un example avec le cas de la *Figure 1* :

```
import Game
```

7.1.2. Importation d'un sous package

Pour importer un sous-package, il faut nommer le package au dessus dans la hiérarchie, suivi du nom du sous-package et séparer les deux noms par un point :

```
import Game.Level
```

7.1.3. Importation d'un module

C'est le même princpe que précédemment pour importer un module : il faut nommer la hiérarchie. Exemple :

```
import Game.Level.start
```

On peut néanmoins utiliser une autre formulation:

```
from Game.Level import start
```

7.1.4. Importation d'une fonction

Pour importer une fonction venant d'un module, il faut d'abord importer le package et le module (voire le sous-package si cela s'y prête). Admettons que dans le module `start.py` il y ait la fonction `select_difficulty()` qui prend comme argument le chiffre 1 pour facile, et 2 pour difficile. Il y a trois façons différentes d'importer et d'utiliser la fonction :

```
# methode 1
import Game.Level.start # importation
Game.Level.start.select_difficulty(2) # utilisation

# methode 2
from Game.Level import start # importation
start.select_difficulty(2) # utilisation
```

```
# methode 3
from Game.Level.start import select_difficulty # importation
select_difficulty(2) # utilisation
```

La première méthode a l'inconvénient de générer des longues commandes lors de l'utilisation de la fonction `select_difficulty()`. La troisième méthode permet d'importer uniquement la fonction voulue du module. Elle n'est pas recommandée car il vaut toujours mieux inclure le nom du module dans la commande (pour éviter des risques de collisions entre fonctions). Avec la librairie PsychoPy, c'est donc plutôt la deuxième méthode que nous allons utiliser. Voici un exemple (avec la fonction `core`) :

```
from psychopy import core # importation
core.wait(1) # utilisation
```

7.1.5. Quoi importer au final ?

La plupart du temps, vous importerez l'entièreté d'un package (plutôt que simplement un module ou une fonction) car il y a de fortes chances pour qu'il y ait plus qu'une fonction qui vous intéresse dans une librairie ou un module.

7.2. Librairies importantes de Python

Certains modules sont dans la Python Standard Library et peuvent donc être appelés sans spéficier le package (car il est importé par défaut). C'est le cas du module 'math' qui contient un ensemble de fonction pour ... faire des opérations mathématiques."

```
import math
# Pour voir le contenu du module:
print dir(math)

> ['__doc__', '__file__', '__name__',
'__package__', 'acos', 'acosh', 'asin',
'asinh', 'atan', 'atan2', 'atanh', 'ceil',
'copysign', 'cos', 'cosh', 'degrees',
'e', 'erf', 'erfc', 'exp', 'expm1',
'fabs', 'factorial',
'floor', 'fmod', 'frexp', 'fsum', 'gamma',
'hypot','isinf', 'isnan', 'ldexp', 'lgamma',
'log', 'log10', 'log1p', 'modf', 'pi', 'pow',
'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh',
'trunc']
```

Vous pouvez voir par vous-même que ce module contient des fonctions mathématiques permettant par exemple de calculer le sinus d'un chiffre :

```
print math.sin(45)
> 0.850903524534
```

La liste des packages et modules cruciaux de Python varie en fonction des utilisateurs (voir ici, ici ou ici).

8. Gestion avancée du timing (Optionnel)

8.1. Parler en frame

Nous l'avons déjà vu (Chapitre 2, partie optionnelle), parler en nombre de frames est la meilleure méthode pour gérer correctement le timing, car elle est précise et sujette à moins d'erreurs. Admettons que votre ordinateur ait un taux de rafraichissement de 60 Hz. Si vous voulez afficher votre stimuli (`o`) pendant 100 msecs, cela correspond à 6 frames. Voici le code à écrire :

```
for frameN in range(6):
  o.draw()
 w.flip()
```

Que se passe-il ? `range(6)` indique que `frameN` va prendre 6 valeurs, donc la boucle va être jouée 6 fois. Au premier passage, l'objet `o` est dessiné dans la fenêtre et affiché. Cela se déroule sur un frame. Comme il y a 6 passages dans la boucle, le tout va durer 6 frames. **Littéralement, PsychoPy ne sait pas que `frameN` correspond à un numéro de frames** (d'ailleurs nous aurions pu remplacer `frameN` par `n`, `i` ou `lapinou`). La raison pour laquelle cela va durer 6 frames, c'est que la boucle va être jouée 6 fois et qu'à chaque fois il faut 'flipper' quelque chose sur l'écran. Or un flip prend un frame. Donc six flips prendront six frames (**si ce qui est à flipper n'est pas trop demandeur en ressources !!**, sinon cela peut durer plus de temps !).

Vous pourriez vouloir vérifier que cette boucle dure bien 100 msecs, et ce serait une bonne idée. Demandons donc à PsychoPy de calculer le temps écoulé entre le début et la fin de la boucle. Voici le même script, avec les modifications nécessaires :

```
01    horloge = core.Clock()
02    for frameN in range(6):
03      o.draw()
04      w.flip()
05    print horloge.getTime()*1000
```

Sur mon ordinateur, voici la réponse :

```
> 95.692873
```

Le résultat obtenu est un peu surprenant : nous attendions à quelque chose proche de 100 ms, or 95 msecs est assez éloigné. Refaisons alors l'expérience 12 fois de suite. Il suffit d'inclure nos lignes de code dans une boucle supérieure :

```
01    for i in range(12):
02      horloge = core.Clock()
03      for frameN in range(6):
04        o.draw()
05        w.flip()
06      print horloge.getTime()*1000
```

Dans la fenêtre Output, j'obtiens :

```
96.8501567841
100.195169449
99.5299816132
100.135803223
100.867033005
99.3590354919
100.675821304
99.2119312286
101.274967194
99.9548435211
100.42309761
99.4579792023
```

Voilà qui est mieux ! On constate qu'il y a une petite imprécision de maximum 1.3 msecs. Cela peut être dû à plusieurs choses : mon ordinateur est connecté à internet, 14 autres applications sont ouvertes et tournent en même temps que PsychoPy,... **Ce serait effectivement une très mauvaise idée de faire son testing dans ces conditions**, car ces applications et internet utilisent des ressources de l'ordinateur qui ne sont donc pas allouées à PsychoPy, pouvant entrainer quelques décalages dans le traitement des commandes.

Dans le manuel de PsychoPy, à la rubrique *2.8 Timing Issues and synchronisation* vous trouverez un ensemble de recommandations si la précision du timing est cruciale pour votre expérience.

Remarquez enfin que l'imprécision est la plus élevée lors du premier passage dans la boucle. C'est quelque chose qui se retrouve dans tous les logiciels de ce type. Cela justifie en partie de toujours utiliser des *restart items* (nous y viendrons dans le dernier chapitre). Lorsqu'il y a une pause dans votre script (= le script s'arrête sur une commande : affichage d'une page de consigne, d'une page de pause, ...), le premier essai qui suit subit toujours une imprécision de timing. **Mettre un 'essai restart' (un stimuli que vous n'incluez pas dans vos analyses) permet autant à Python qu'à votre participant de se 'remettre' dans le bain**.

Pour revenir à la question des frames, je vous conseille dans tous les cas de prendre l'habitude de parler en frames.

Admettons que vous ayez 2 stimuli à présenter à la suite (o1 et o2), le premier pendant 6 frames (100 ms) et le second pendant 12 frame (200 ms), voici une façon de faire :

```
01    for frameN in range (18):
02      if 0<=frameN<12:
03        o1.draw()
04      if 12<=frameN<18:
05        o2.draw()
06      w.flip()
```

8.2. Attendre une réponse un certain temps

Nous avons vu que si nous voulons une réponse par appui clavier dans un certain délail, on peut écrire les commandes suivantes :

```
event.clearEvents()


o.draw()
w.flip()
event.waitKeys(maxWait=2, keyList=['y','o'])
```

Dans ce cas cependant, nous somme obligés de parler en secondes, ce qui n'est pas optimal. Pour utiliser les frames, il nous faut écrire les commandes avec 'getKeys()' :

```
event.clearEvents()
for frameN in range(120):
  o.draw()
  w.flip()
  if event.getKeys('y'):
    break
```

Un appui sur la touche Y est attendu pendant maximum 120 frames (2 secs sur un ordinateur 60 Hz).

Le même principe est à utiliser pour la souris. Au lieu d'utiliser un objet lié à l'horloge interne, on définit par exemple l'attente d'un clic de souris pendant une certaine période :

```
maSouris = event.Mouse()
for frameN in range(120):
  o.draw()
  w.flip()
  if maSouris.getPressed()[0]==1:
    break
```