1 Introduction

Dans la vie courante, on est souvent confronté à des choix et on choisit de faire telle ou telle chose suivant les circonstances que l'on observe :

- **Exemple 1 :** *je dois aller chercher du pain mais s'il pleut, alors je prends mon parapluie.*
  Je regarde dehors :
  - Mardi, il pleut donc je prends mon parapluie, puis je vais chercher mon pain.
  - Mercredi, il fait beau : je vais chercher mon pain.

- **Exemple 2 :** *je dois faire un gâteau. Si dans mon réfrigérateur j'ai au moins quatre oeufs, alors je fais un gâteau au chocolat ; sinon, je fais une tarte aux fraises.*
  Je regarde dans mon frigo :
  - Mardi, il y a sept oeufs, donc je fais mon gâteau au chocolat.
  - Jeudi, il y a trois oeufs donc je fais une tarte aux fraises.

En algorithmique et programmation, on observe le même type de phénomène :

si une condition donnée est vraie, alors je vais réaliser telles instructions ; mais, si elle est fausse, alors je poursuis ce que j'ai à faire ou je réalise d'autres instructions (qui peuvent elles-mêmes dépendre d'une nouvelle condition).

Cette structure algorithmique est appelée **structure d'embranchement**.

2 Conditions et tests

Définition 2.1 (Condition) Une *condition* est un énoncé qui peut prendre l'une des deux valeurs suivantes : **Vrai** ou **Faux**.

On dit également qu'une condition est une valeur de type **logique** ou de type **booléen** (`bool` en `Python`).

Une condition est réalisée à l'aide de **tests** et d'**opérateurs logiques**.

2.1 Les tests

La plupart des tests est réalisée à l'aide d'**opérateurs de comparaison**. Le tableau ci-dessous regroupent les opérateurs usuels pour les types numériques entiers et flottants.

égal	différent	inférieur	supérieur	inférieur strict	supérieur strict
==	!=	<=	>=	<	>

Exemple 2.2

- `2 == 3` renvoie `False` et `2 != 3` renvoie `True`
- `2 > 3` renvoie `False` et `2 < 3` renvoie `True`
- `2 >= 3` renvoie `False` et `2 <= 3` renvoie `True`

Remarque 2.3 Il existe d'autres opérateurs de comparaison, comme `in`, que l'on peut utiliser lors de la manipulation de boucles, de listes ou de chaînes de caractères (voir les chapitres 5, 6 et 7).

Remarque 2.4 Il existe également des fonctions natives en `Python` qui réalise des tests. Par exemple, la fonction `isinstance` est une fonction qui renvoie `True` si la valeur de son premier paramètre est du type donné par son second paramètre, et `False` sinon. Ainsi, l'instruction `isinstance(3,int)` renvoie `True` et l'instruction `isinstance(3.0,int)` renvoie `False`.

On peut bien sûr écrire ses propres fonctions tests (voir l'exercice 2.6 ci-dessous).

2.2 Les opérateurs logiques

Un opérateur logique est un opérateur s'appliquant à des conditions et renvoyant une condition.

Les trois opérateurs logiques classiques sont les suivants :

- le **non** (mot-clé `not` en `Python`) qui permet de faire la négation d'une condition ;
- le **ou** (mot-clé `or` en `Python`) qui renvoie Vrai quand au moins une des deux conditions est vraie, et Faux sinon ;
- le **et** (mot-clé `and` en `Python`) qui renvoie Vrai quand les deux conditions sont vraies simultanément, et Faux sinon.

Ces trois opérateurs sont définis de façon logique par leur table de vérité :

condition P	condition Q	condition non(P)	condition P ou Q	condition P et Q
Vrai	Vrai	Faux	Vrai	Vrai
Vrai	Faux	Faux	Vrai	Faux
Faux	Vrai	Vrai	Vrai	Faux
Faux	Faux	Vrai	Faux	Faux

Exemple 2.5 Soit `a` la variable définie par `a = 4`. Alors :

- `a%2 == 0` renvoie `True`
- `not(a%2 == 0)` renvoie `False`
- `isinstance(a,str)` renvoie `False`
- `not(isinstance(a,str))` renvoie `True`
- `a >= 0 and a < 5` renvoie `True`
- `not(a >= 0 and a < 5)` renvoie `False`
- `a >= 0 and a < 4` renvoie `False`
- `a >=0 or a < 4` renvoie `True`
- `isinstance(a,int) and a >= 0` renvoie `True`
- `isinstance(a,float) and a >= 0` renvoie `False`
- `isinstance(a,float) or a >= 0` renvoie `True`

2.3 Exercices

Exercice 2.6 Compléter le Notebook *NSI Première Partie 1 Chapitre 4 Conditions et tests*.

Exercice 2.7 (QCM)

1. Une variable `a` est définie et on saisit l'instruction `a == 4`. Parmi les affirmations suivantes, lesquelles sont vraies ?
   (a) on affecte la valeur `4` à la variable `a`
   (b) on teste si la valeur contenue dans `a` est égale à `4`
   (c) si `a` contient la valeur `'4'`, alors le résultat est `True`
   (d) si `a` contient la valeur `4`, alors le résultat est `True`

2. Une variable `a` est définie et on saisit l'instruction `(a >= 3 and a%2 == 0)`. Le résultat est `True` lorsque :
   (a) la variable `a` contient la valeur `2`
   (b) la variable `a` contient un nombre pair
   (c) la variable `a` contient la valeur `4`
   (d) la variable `a` contient la valeur `5`

3. On saisit les instructions suivantes :

```
a = 3
b = a + 1
a = (a + 5) // 2
```

   Parmi les instructions suivantes, lesquelles renvoient `True` ?
   (a) `a == b` (b) `a != b` (c) `not(a >= 3)` (d) `not(a%2 == 1)`

4. On saisit les instructions suivantes :

```
a = 3
b = 2*a + 1
```

   Parmi les instructions suivantes, lesquelles renvoient `True` ?
   (a) `b%2 == a%2`
   (b) `b >= 0 and b < 10`
   (c) `b >= a or b <= 0`
   (d) `a%2 == 1`

5. `a`, `b` et `c` sont trois variables booléennes. Parmi les affirmations suivantes, lesquelles sont vraies ?
   (a) `(a and b) or c == (a or c) and (b or c)`
   (b) `not(a or b) == not(a) and not(b)`
   (c) `not(a and b) == not(a) or not(b)`
   (d) `(a or b) and c == (a and c) or (b and c)`

3 Les structures d'embranchement

Définition 3.1 Les *structures d'embranchement* sont des structures algorithmiques qui permettent de réaliser des instructions différentes suivant les valeurs de conditions préalablement vérifiées.

Il existe deux types de structures d'embranchement :

- les **embranchements simples** qui utilisent une seule condition ;
- les **embranchements multiples** qui utilisent plusieurs conditions.

3.1 Les embranchements simples

Les embranchements simples peuvent être avec ou sans alternative.
On donne ci-dessous leur structures algorithmiques et leur implémentations `Python`.

3.1.1 Embranchement simple sans alternative

```
si condition vraie alors
   bloc d'instructions
fin si
```

```
if condition vraie:      # ne pas oublier les :
   bloc d'instructions   # indentation obligatoire
# pour terminer, on stoppe l'indentation
```

Exemple 3.2 La fonction `estPositif` ci-dessous renvoie `True` si la valeur passée en paramètre est positive ou nulle, et renvoie `False` si elle est strictement négative. On donne à la fois la version algorithmique et son implémentation `Python`, ainsi que deux exemples d'appel pour comprendre comment fonctionne la structure d'embranchement `si ... alors ... fin si`.

Pseudo-code :

```
fonction estPositif(x) ①
   si x >= 0 alors ②
      renvoyer Vrai ③
   fin si
   renvoyer Faux ④
fin fonction
```

`Python` :

```
def estPositif(x): ①
   if x >= 0: ②
      return True ③
   return False ④
```

- Que se passe-t-il lors de l'appel `estPositif(5)` ?
  - ① Le paramètre `x` prend la valeur `5`
  - ② Exécution du test `x >= 0` : la réponse est `True` donc la condition est vraie
  - ③ On entre dans le `alors` et la fonction renvoie `True`

- Que se passe-t-il lors de l'appel `estPositif(-3)` ?
  - ① Le paramètre `x` prend la valeur `-3`
  - ② Exécution du test `x >= 0` : la réponse est `False` donc la condition est fausse
  - ④ On sort de la structure `si ... alors ... fin si` et la fonction renvoie `False`

3.1.2 Embranchement simple avec alternative

```
si condition vraie alors
   bloc d'instructions 1
sinon
   bloc d'instructions 2
fin si
```

```
if condition vraie:      # ne pas oublier les :
   bloc d'instructions 1 # indentation obligatoire
else:                    # ne pas oublier les :
   bloc d'instructions 2 # indentation obligatoire
# pour terminer, on stoppe l'indentation
```

Exemple 3.3 La fonction `signe` ci-dessous donne le signe au sens large de la valeur passée en paramètre. On donne à la fois la version algorithmique et son implémentation `Python`, ainsi que deux exemples d'appel pour comprendre comment fonctionne la structure d'embranchement `si ... alors ... sinon ... fin si`.

Pseudo-code :

```
fonction signe(x) ①
   si x >= 0 alors ②
      renvoyer 'positif' ③
   sinon
      renvoyer 'négatif' ④
   fin si
fin fonction
```

Python :

```
def signe(x): ①
   if x >= 0: ②
      return 'positif' ③
   else:
      return 'négatif' ④
```

⋆ Que se passe-t-il lors de l'appel `signe(5)` ?

① Le paramètre `x` prend la valeur `5`

② Exécution du test `x >= 0` : la réponse est `True` donc la condition est vraie

③ On entre dans le `alors` et la fonction renvoie `'positif'`

⋆ Que se passe-t-il lors de l'appel `signe(-3)` ?

① Le paramètre `x` prend la valeur `-3`

② Exécution du test `x >= 0` : la réponse est `False` donc la condition est fausse

④ On entre dans le `sinon` et la fonction renvoie `'négatif'`

3.2 Les embranchements multiples

Les embranchements multiples utilisent plusieurs conditions successives (au moins deux) pour savoir dans quel cas on se situe. On a donc des successions de `sinon si`.
On donne ci-dessous la structure algorithmique et l'implémentation `Python` correspondant à un embranchement multiple avec deux conditions.

```
si condition 1 vraie alors
   bloc d'instructions 1
sinon si condition 2 vraie alors
   bloc d'instructions 2
sinon
   bloc d'instructions 3
fin si
```

```
if condition 1 vraie:       # ne pas oublier les :
   bloc d'instructions 1    # indentation obligatoire
elif condition 2 vraie:     # ne pas oublier les :
   bloc d'instructions 2    # indentation obligatoire
else:                       # ne pas oublier les :
   bloc d'instructions 3    # indentation obligatoire
# pour terminer, on stoppe l'indentation
```

Remarque 3.4 Comme pour les embranchements simples, l'alternative `sinon` est optionnelle.

Exemple 3.5 La fonction `signeStrict` ci-dessous donne le signe au sens strict de la valeur passée en paramètre. On donne à la fois la version algorithmique et son implémentation `Python`, ainsi que trois exemples d'appel pour comprendre comment fonctionne la structure d'embranchement `si ... alors ... sinon si ... alors... sinon ... fin si`.

Pseudo-code :

```
fonction signeStrict(x) ①
   si x > 0 alors ②
      renvoyer 'strictement positif' ③
   sinon si x = 0 alors ④
      renvoyer 'nul' ⑤
   sinon
      renvoyer 'strictement négatif' ⑥
   fin si
fin fonction
```

Python :

```
def signeStrict(x): ①
   if x > 0: ②
      return 'strictement positif' ③
   elif x == 0: ④
      return 'nul' ⑤
   else:
      return 'strictement négatif' ⑥
```

⋆ Que se passe-t-il lors de l'appel `signeStrict(5)` ?

① Le paramètre `x` prend la valeur `5`

② Exécution du test `x > 0` : la réponse est `True` donc la condition est vraie

③ On entre dans le `alors` et la fonction renvoie `'strictement positif'`

⋆ Que se passe-t-il lors de l'appel `signeStrict(0)` ?

① Le paramètre `x` prend la valeur `0`

② Exécution du test `x > 0` : la réponse est `False` donc la condition est fausse

④ On passe au `sinon si` et à l'exécution du test `x = 0` : la réponse est `True` donc la condition est vraie

⑤ On entre dans le `alors` et la fonction renvoie `'nul'`

⋆ Que se passe-t-il lors de l'appel `signe(-3)` ?

① Le paramètre `x` prend la valeur `-3`

② Exécution du test `x > 0` : la réponse est `False` donc la condition est fausse

④ On passe au `sinon si` et à l'exécution du test `x = 0` : la réponse est `False` donc la condition est fausse

⑥ On entre dans le `sinon` et la fonction renvoie `'strictement négatif'`

4 Exercices

Exercice 4.1 (Embranchements simples) Compléter le Notebook *NSI Première Partie 1 Chapitre 4 Embranchements simples*.

Exercice 4.2 (Embranchements multiples) Compléter le Notebook *NSI Première Partie 1 Chapitre 4 Embranchements multiples*.

Exercice 4.3 (QCM)

1. On considère la fonction suivante :

```
def prix(n):
    if n <= 5:
        p = 2*n
    else:
        p = 1.8*n
    return p
```

Parmi les affirmations suivantes, lesquelles sont vraies ?

(a) `prix(5)` renvoie `10`

(b) `prix(2)` renvoie `3.6`

(c) `prix(5)` renvoie `9.0`

(d) `prix(8)` renvoie `14.4`

2. On considère la fonction suivante :

```
def dist(a,b):
    if a > b:
        return a - b
    return b - a
```

Parmi les affirmations suivantes, lesquelles sont vraies ?

(a) `dist(3,2)` renvoie `1`

(b) le code n'est pas correct

(c) `dist(2,3)` renvoie `-1`

(d) `dist(1,1)` renvoie `0`

3. On considère la fonction suivante :

```
def mystere(a):
    if isinstance(a,int):
        return a
    return False
```

Parmi les affirmations suivantes, lesquelles sont vraies ?

(a) le code est faux : il n'y a pas de tests après le `if`

(b) `mystere('3')` renvoie `'3'`

(c) `mystere(3)` renvoie `3`

(d) la fonction `mystere` renvoie `False` dès que son paramètre n'est pas entier

4. On considère la fonction suivante :

```
def syr(n):
    if n%2 == 0:
        return n//2
    else:
        return 3*n + 1
```

Parmi les affirmations suivantes, lesquelles sont vraies ?

(a) `syr(7)` renvoie `22`
(b) `syr(8)` renvoie `4`
(c) le code n'est pas correct car il y a deux `return`
(d) `syr(8)` renvoie `4.0`

5. On considère la fonction suivante :

```
def calcul(a,b):
    if a >= 3 and a%b == 0:
        return a//b
    return a%b
```

Parmi les affirmations suivantes, lesquelles sont vraies ?

(a) `calcul(1,2)` renvoie `1`
(b) `calcul(5,2)` renvoie `1`
(c) `calcul(8,4)` renvoie `0`
(d) `calcul(8,4)` renvoie `2`

5 Exercices complémentaires

5.1 Présentation de la bibliothèque `turtle`

Ce paragraphe présente les différentes fonctionnalités de la bibliothèque `turtle`. Celle-ci permet de réaliser des dessins géométriques correspondant à la trace d'une petite « tortue » virtuelle (représentée par un petite flèche) se déplaçant dans une fenêtre graphique. Le déplacement de la tortue sur l'écran est contrôlé par des instructions simples issues de la bibliothèque `turtle`. Pour charger cette bibliothèque, n'oubliez pas de commencer votre script par l'instruction

```
from turtle import *
```

ou

```
import turtle
```

en précédant les appels aux fonctions de la bibliothèque `Turtle` par `turtle.`.

Remarque 5.1 Si on travaille sous Jupyter, il faut terminer son code par l'instruction `done()`.

5.1.1 Manipulation de la fenêtre graphique

- `setup(W,H)` : initialise la fenêtre graphique ; ses deux arguments représentent respectivement sa largeur et sa hauteur. L'origine $(0;0)$ est situé au centre de la fenêtre et la tortue est placée initialement à cet endroit avec une orientation vers la droite (*n'existe pas sous Jupyter*).
- `window_width()` et `window_height()` : renvoient respectivement la largeur et la hauteur de la fenêtre graphique.
- `bgcolor(c)` : donne la couleur `c` à l'arrière-plan de la fenêtre graphique ; la couleur `c` est une couleur prédéfinie donnée sous forme de chaîne de caractères (voir tableau ci-dessous). On peut également donner les couleurs sous la forme d'un triplet `(r,g,b)` en indiquant la commande `colormode(255)`.
- `reset()` : efface le contenu de la fenêtre graphique et replace la tortue en son centre.
- `clear()` : efface le contenu de la fenêtre graphique mais conserve la tortue en sa dernière position.
- `title(name)` : donne comme nom à la fenêtre graphique la chaîne de caractères `name`.
- `exitonclick()` : quitte la fenêtre graphique en cliquant dedans (*n'existe pas sous Jupyter*).
- `getcanvas().postscript(file="monoeuvre.eps")` : permet de sauvegarder le dessin obtenu dans un fichier `.eps` (appelé ici, `monoeuvre.eps`) de façon à pouvoir l'éditer ultérieurement, par exemple avec `HTML`.

5.1.2 Gestion de la tortue

- `goto(x,y)` : amène la tortue au point de coordonnées `(x;y)`.
- `forward(d)` ou `fd(d)` : avance la tortue d'une distance `d` (par rapport à l'orientation).
- `backward(d)` ou `bk(d)` : recule la tortue d'une distance `d` (par rapport à son orientation).
- `position()` ou `pos()` : renvoie la position de la tortue.
- `home()` : replace la tortue au centre de la fenêtre graphique.
- `distance(x,y)` : renvoie la distance entre la tortue et le point de coordonnées `(x;y)`.
- `left(a)` ou `lt(a)` : pivote la tortue vers la gauche d'un angle `a` (exprimé en degrés).
- `rigth(a)` ou `rt(a)` : pivote la tortue vers la droite d'un angle `a` (exprimé en degrés).
- `setheading(a)` : oriente la tortue d'un angle `a` (en degrés) ; en particulier, si `a=0`, la tortue est orientée vers la droite, si `a=90`, elle est orientée vers le haut, si `a=180`, elle est orientée vers la gauche et si `a=270`, elle est orientée vers le bas.
- `heading()` : renvoie l'orientation de la tortue.
- `hideturtle()` ou `ht()` : rend la tortue invisible (accélère le tracé).
- `showturtle()` ou `st()` : rend la tortue visible.

5.1.3 Tracé de formes

- `up()` ou `penup()` ou `pu()` : relève la tortue (pour pouvoir se déplacer sans dessiner).
- `down()` ou `pendown()` ou `pd()` : abaisse la tortue (pour recommencer à dessiner).
- `circle(r)` : trace un cercle de rayon `r` ; le tracé commence à l'endroit où est la tortue, dans la direction de la tortue puis tourne sur la gauche.
- `circle(r,a)` : trace un arc de cercle de rayon `r` et d'angle `a` (exprimé en degrés).
- `color(c)` : donne la couleur `c` au trait.
- `color(c,d)` : donne la couleur `c` au trait et remplit la figure avec la couleur `d`.
- `begin_fill() ... end_fill()` : remplit un contour à l'aide de la couleur sélectionnée ; les `...` doivent être remplacés par une suite d'instructions définissant une forme géométrique.
- `width(e)` : donne l'épaisseur `e` au trait.
- `shape(f)` : change la forme de la tortue, `f` est une chaîne de caractères (`"turtle"`, `"circle"`, `"arrow"` (par défaut), `"square"`, `"triangle"`, `"classic"`, `"blank"`)
- `speed(v)` : règle la vitesse `v` de tracé (`"slowest"`, `"slow"`, `"normal"` (par défaut), `"fast"`, `"fastest"`).
- `write(text)` : écrit la chaîne de caractères `text` dans la fenêtre graphique.

5.1.4 Couleurs prédéfinies

Les couleurs prédéfinies sont données sous la forme de chaîne de caractères. Par exemple, "`black`" donne la couleur noire et `"white"` la couleur blanche. Pour les autres couleurs, on a le tableau de correspondance suivant :

bleu	rouge	vert	jaune	marron	rose	orange	violet	gris
`"blue"`	`"red"`	`"green"`	`"yellow"`	`"brown"`	`"pink"`	`"orange"`	`"purple"`	`"grey"`

5.2 Exercices

Exercice complémentaire 5.2 (Un exemple de tracé avec `turtle`) Saisir et interpréter chaque ligne du programme suivant.

```
from turtle import *

def exempleTrace():
    up() ; setheading(90) ; goto(0 , -300) ; down()
    color('yellow')
    goto(0, 300)
    up() ; goto(-175, 0) ; down()
    color('blue', 'blue')
    begin_fill() ; circle(50) ; end_fill()
    up() ; goto(275, 0) ; down()
    color('red', 'green')
    begin_fill() ; circle(50) ; end_fill()

exempleTrace()
done()
```

Exercice complémentaire 5.3 (Contrôle de distance)

1. (a) Saisir la fonction suivante et expliquer à quoi elle sert :

```
from turtle import *

def droite(d):
    maxX= window_width()/2
    x, y = position()
    if x + d < maxX:
        goto(x + d, y)
```

   (b) Ecrire sur le même modèle une fonction `bas(distance)` déplaçant la tortue verticalement en vérifiant qu'elle ne sort pas de la fenêtre graphique.

2. (a) Saisir la fonction suivante et expliquer à quoi elle sert :

```
from turtle import *

def horizontal(distance):
    if distance > 0:
        forward(distance)
    else:
        backward(-distance)
```

   (b) Ecrire sur le même modèle une fonction `vertical(distance)` déplaçant la tortue verticalement en vérifiant qu'elle ne sort pas de la fenêtre graphique.