Premiers pas avec Python

Objectifs :

- Pourquoi programmer ?
- Prise en main d'un IDE
- Les variables
- Les différents types de variable
- Les conditions
- Les boucles FOR et WHILE
- Utilisation de bibliothèques comme math et turtle

Programmer un ordinateur, c'est quoi ?

Un programme est la description d'un algorithme dans un langage compréhensible par un humain et par une machine qui l'exécute afin de traiter des données.

Programmer, c'est donner des suites d'instructions à un l'ordinateur, un ordinateur sans programme ne sait rien faire.

Il existe différents langages qui permettent de programmer un ordinateur, mais le seul directement utilisable par le processeur est le langage machine (suite de 1 et de 0), aussi appelé binaire. Aujourd'hui (presque) plus personne ne programme en binaire (trop compliqué).

Les informaticiens utilisent des instructions (mots souvent en anglais) en lieu et place de la suite de 0 et de 1. Ces instructions, une fois écrites par le programmeur, sont « traduites » en langage machine. Un programme spécialisé assure cette traduction. Ce système de traduction s'appellera `interpréteur` ou bien `compilateur`, suivant la méthode utilisée pour effectuer la traduction.

Il existe 2 grandes familles de langages de programmation :

- Les langages de bas niveau sont très complexes à utiliser, car très éloignés du langage naturel, on dit que ce sont des langages « proches de la machine », en contrepartie ils permettent de faire des programmes très rapides à l'exécution. L'assembleur est le

langage de bas niveau. Certains "morceaux" de programmes sont écrits en assembleur encore aujourd'hui.

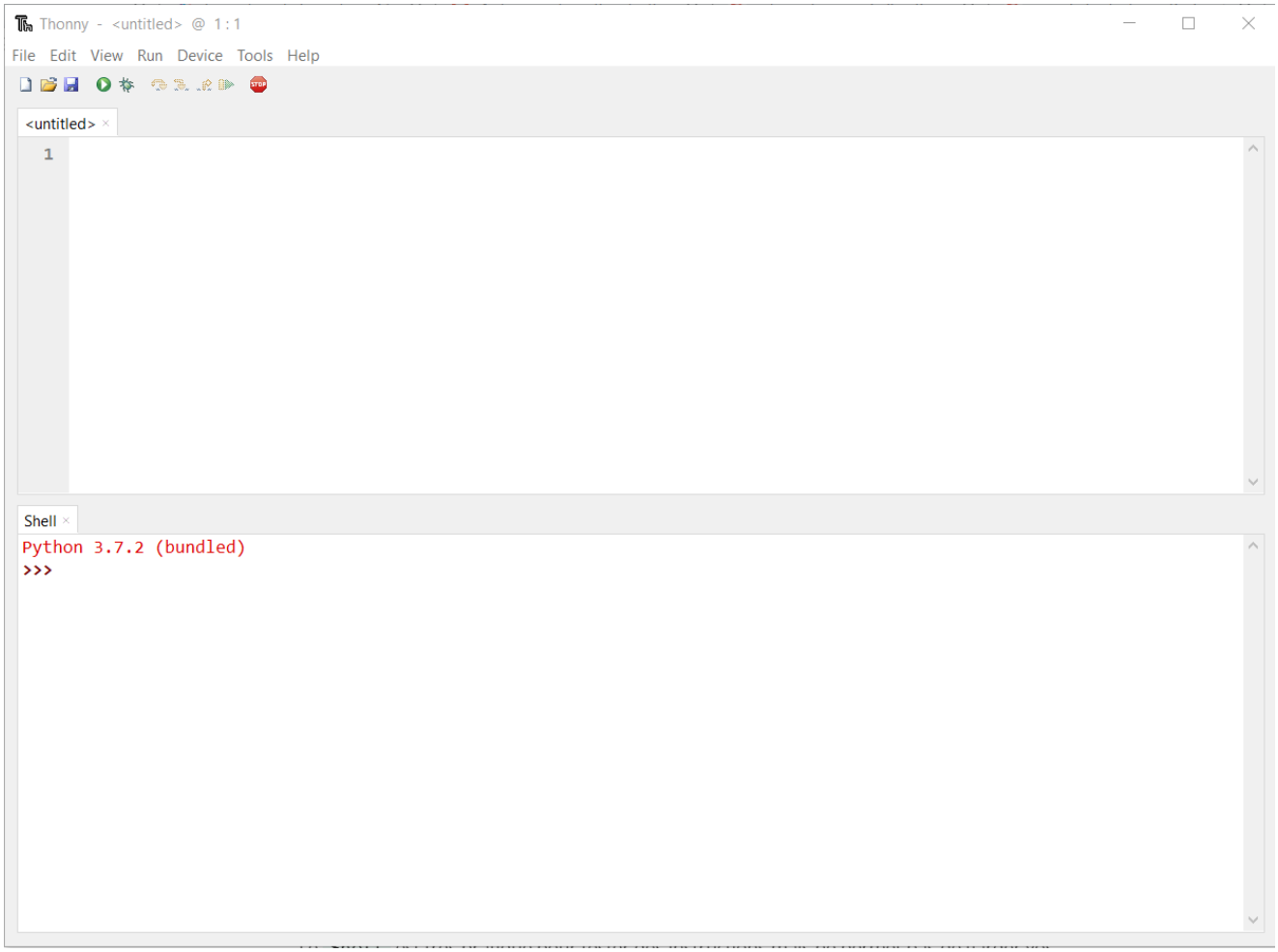
- Les langages de haut niveau sont eux plus « faciles » à utiliser, car plus proches du langage naturel. Exemples de langages de haut niveau : C, C++ , java, Qbasic, Python...

Cette année, nous allons apprendre les bases de la programmation en utilisant le langage nommé Python.

Votre environnement de travail

Utilisation de Thonny

Une fois l'application Thonny lancé, vous obtenez :



On observe une fenêtre avec sa barre de menus usuels, une rangée de boutons, et deux panneaux correspondant à :

- l'éditeur : onglet nommé (pour l'instant) `<untitled>`
- l'interpréteur : onglet nommé `Shell`

La zone `<untitled>` est une zone d'édition dans laquelle vous pourrez rédiger votre code et le sauvegarder : c'est le `mode programme`.

La zone `shell` est celle où vous pouvez exécuter des commandes Python, et obtenir directement le résultat à l'image d'une calculatrice : c'est le `mode interactif`.

Remarque : La version de Python utilisée `Python 3.7.2`

Ce sont ces deux zones que vous utiliserez essentiellement lors de tous vos travaux d'informatique par un va et vient entre ces deux zones : la zone d'édition pour rédiger/corriger/améliorer votre code, et le shell pour tester et valider votre code.

Traditionnellement, les "apprentis programmeurs" commencent leur "carrière" en écrivant un programme qui permet d'afficher à l'écran le message "Hello World !". Nous n'allons pas déroger à cette tradition. En Python, il suffit d'une "instruction" pour afficher ce message (notez bien que selon le langage utilisé cela peut-être plus complexe).

Dans la zone du `shell`, il y a trois chevrons qui constituent l'invite de commandes avec lesquels nous saisissons les instructions ci-dessous :

```
>>> print("Hello World !")

# on obtient :
Hello World !
```

Remarque : dans le `shell`, si vous désirez rappeler l'instruction saisie précédemment, pas besoin de tout retaper, il suffit d'utiliser la flèche haute du clavier pour retrouver le code précédent et de revalider (après une éventuelle modification).

Le `shell` est très pratique pour tester des instructions mais ne permet pas de garder vos programmes pour les réutiliser plus tard, pour cela il faut saisir son code dans la fenêtre nommée `<untitled>` pour le moment.

Dans la fenêtre nommée `<untitled>`, nous saisissons l'instruction ci-dessous :

```
print("Hello World !")

# on obtient :
Hello World !
```

Pour démarrer le programme, il faut préalablement le sauvegarder (en lui donnant un nom de la forme `programme.py`) puis le lancer à l'aide de l'icône en forme de flèche (ou utilisez `F5`).

On obtient alors le résultat dans la fenêtre du `Shell`.

Remarque : il est possible d'utiliser '...' à la place de "..." autour du texte à afficher dans l'instruction `print`.

Utilisation d'un éditeur en ligne

Il est parfois commode d'utiliser un éditeur en ligne utilisable dans un simple navigateur.

Après avoir ouvert votre navigateur internet de votre choix, tapez par exemple l'adresse suivante dans la barre d'adresse de ce dernier : http://pythonfiddle.com/

Dans la fenêtre principale, saisissez l'instruction ci-dessous, puis cliquez ensuite sur le bouton "Run" (en haut à gauche)

```
print("hello world !")
```

Si tout s'est bien passé vous devriez avoir le message "Hello World !" qui s'affiche dans une fenêtre qui se situe en bas de l'écran.

Il existe de nombreux IDE en ligne : https://codeboard.io, https://repl.it/, https://trinket.io/python, http://pythontutor.com/

Les éditeurs en ligne peuvent être utiles sur de petits projets ou pour travailler de manière collaborative.

Notion de variable

Définition du mot ordinateur d'après "Le Petit Larousse" : « Machine automatique de traitement de l'information, obéissant à des programmes formés par des suites d'opérations arithmétiques et logiques. »

Qui dit "traitement de l'information", dit donc données à manipuler. Un programme "passe" donc son temps à traiter des données. Pour pouvoir traiter ces données, l'ordinateur doit les ranger dans sa mémoire (RAM - Random Access Memory). La RAM se compose de cases dans lesquelles nous allons ranger ces données (une donnée dans une case). Chaque case a une adresse (ce qui permet au processeur de savoir où sont rangées les données).

Alors, qu'est-ce qu'une variable ?

Eh bien, c'est une petite information (une donnée) temporaire que l'on stocke dans une case de la RAM. On dit qu'elle est "variable" car c'est une valeur qui peut changer pendant le déroulement du programme.

Une variable n'est rien d'autre qu'un nom que l'on donne à un objet pour se souvenir de lui et le manipuler.

Une variable est constituée de 2 choses :

- Elle a une `valeur` : c'est la donnée qu'elle stocke (par exemple le nombre 5)

- Elle a un `nom` : c'est ce qui permet de la reconnaître. Nous n'aurons pas à retenir l'adresse de mémoire, nous allons juste indiquer des noms de variables à la place.

```
>>> i = 12
```

Grâce à cette ligne, nous avons défini une variable qui porte le nom (ou identificateur) `i` et qui «contient» la valeur 12 : on parle d'`affectation`.

A tout moment, on peut accéder à la valeur mémorisée :

```
>>> i
12
```

Comme vous pouvez le constater, il suffit d'indiquer le nom de la variable dans le shell et de valider pour obtenir sa valeur.

Affichage d'une variable avec l'instruction print

Par contre, le lancement en mode programme d'une affectation de variable n'affiche rien :

```
point_de_vie = 100

# on obtient
...rien...
```

Il faut ajouter une instruction pour demander l'affichage de la variable :

```
point_de_vie = 100
print(point_de_vie)

# on obtient
100
```

Remarque : l'instruction `print` permet d'afficher la valeur d'une variable, mais attention, si on écrit :

```
point_de_vie = 100
print("point_de_vie")

# on obtient
point_de_vie
```

Allez plus loin avec l'instruction print

On peut facilement afficher du texte associé à la valeur d'une ou plusieurs variables, voici quelques exemples :

```
score = 32
print("Votre score est",score)

# on obtient
Votre score est 32
```

```
age = 18
print("Paul a ",age," ans")

# on obtient
Paul a 18 ans
```

Remarque : il existe des arguments qui permettent de remplacer les espaces mis automatiquement, d'empêcher le passage à la ligne mais aussi d'autres méthodes plus poussées de formatage de chaîne de caractères que nous étudierons plus tard.

Autre remarque : pour le moment, nos variables ont permis de stocker des nombres entiers, nous verrons ensuite qu'il existe d'autres types de variables.

Affectation simultanée de variables

On peut affecter plusieurs variables en une seule ligne, quelques exemples :

```
# La même valeur pour plusieurs variables
>>> a = b = c = 10
>>> print(a, b, c)
10 10 10

# Des valeurs différentes pour des variables différentes
>>> a, b = 3, 4
>>> print(a, b)
3 4

# Intervertir des variables (sans en utiliser une 3ème)
>>> a, b = b, a
>>> print(a, b)
4 3
```

Quelques conseils pour nommer vos variables

- Un nom de variable doit **obligatoirement** commencer par une lettre, ou bien par le symbole *underscore* (le underscore `_` sous la touche `8` des claviers français).

- Donnez à vos variables un nom qui décrit leur rôle plutôt que leur nature : `somme`, `produit`, `prix`, `pt_de_vie` sont mieux que `nombre` ou `entier`.
- Même si Python 3 les accepte, interdisez-vous les caractères accentués. Contentez-vous de symboles simples. Il vaut mieux nommer une variable `prenom` plutôt que `prénom`, ou `coeur` plutôt que `cœur`.
- À quelques exceptions courantes près (du style `x` et `y` pour des coordonnées), il est préférable de choisir des noms qui font au moins 3 caractères (`nom` est préférable à `nm`, `option` est préférable à `op`)
- Et dans tous les cas, n'appelez **jamais** une variable `l` (L minuscule). Suivant la police de caractères utilisée, il est beaucoup trop facile de confondre le L minuscule avec le chiffre *1* (un) ou le *i* majuscule.
- Une variable devrait tout le temps être écrite en lettres minuscules quitte à séparer les mots d'un underscore (préférez `pos_joueur` à `PosJoueur`)
- Attention, Python est sensible à la casse `pos_joueur` est different de `Pos_Joueur`.
- Certains mots-clés de Python sont *réservés*, c'est-à-dire que **vous ne pouvez pas créer des variables portant ce nom**. En voici la liste pour Python 3 :

  ```
  and del from None True as elif global nonlocal try assert else if not while break except import or with class False in pass yield continue finally is raise def for lambda return
  ```

Un peu de calculs

Un ordinateur est bien évidemment capable d'effectuer des opérations mathématiques (arithmétiques). Les signes utilisés sont classiques : +, - , * (multiplication), / (division), // (division euclidienne), % (modulo : reste d'une division euclidienne) ou ** (exposant)

Il est tout à fait possible d'effectuer des opérations directement avec des nombres, mais il est aussi possible d'utiliser des variables.

Utilisation de variables pour réaliser un calcul

Le programme suivant permet de faire l'addition de deux entiers stockés dans deux variables `a` et `b` en plaçant le résultat dans une troisième variable `resultat` qui est ensuite affichée :

```
a = 12
b = 54
resultat = a + b
print(resultat)

# on obtient
66
```

Modification d'une variable

Soit le programme suivant :

```
a = 11
print(a)
a = a + 1
print(a)

# on obtient
11
12
```

Détaillons-le déroulé du programme :

- nous créons une variable `a` et nous lui attribuons la valeur 11
- nous affichons à l'écran la valeur de `a` (c'est à dire 11)

La suite est un peu plus complexe, mais très importante à comprendre. Il va falloir lire la ligne `a = a + 1` de droite à gauche, décortiquons cette ligne :

- `a + 1` : nous prenons la valeur actuelle de a (c'est-à-dire 11) et nous lui ajoutons 1, à droite de l'égalité nous avons donc maintenant la valeur 12
- nous attribuons la valeur qui vient d'être calculée à la variable a (donc maintenant a = 12)
- nous affichons à l'écran la nouvelle valeur de a

Ce raisonnement peut être généralisé pour éviter des erreurs parfois difficiles à corriger : dans une égalité, commencer toujours par évaluer l'expression se trouvant à droite du signe égal.

Remarque : quand on ajoute un entier (souvent de valeur 1) à une variable, on parle `d'incrémentation`, il existe des raccourcis d'écriture :

```
a = 25
a += 1        # équivalent à : a = a + 1
print(a)

# on obtient
26
```

Quelques exemples avec les autres opérateurs arithmétiques :

```
>>> a = 15
>>> b = 4

>>> a + b    # addition

19
>>> a - b    # soustraction

11
>>> a * b    # multiplication

60
>>> a / b    # division

3.75
>>> a // b    # quotient de la division euclidienne

3
>>> a % b    # modulo = reste de la division euclidienne

3
>>> a ** b    # exposant

50625
```

Règles de priorité des opérateurs

Lorsqu'il y a plus d'un opérateur dans une expression, l'ordre dans lequel les opérations doivent être effectuées dépend de règles de priorité. Sous Python, les règles de priorité sont les mêmes que celles qui vous ont été enseignées au cours de mathématique.

Un moyen mnémotechnique pour les mémoriser, l'acronyme `PEMDAS` :

- `P` pour parenthèses, ce sont elles qui ont la plus haute priorité. Elles vous permettent donc de « forcer » l'évaluation d'une expression dans l'ordre que vous voulez.

  Ainsi `2 * (3 - 1) = 4`, et `(1 + 1) ** (5 - 2) = 8`.

- `E` pour exposants, les exposants sont évalués ensuite, avant les autres opérations.

  Ainsi `2 ** 1 + 1 = 3` (et non 4), et `3 * 1 ** 10 = 3` (et non 59049).

- `M` et `D` pour multiplication et division, qui ont la même priorité. Elles sont évaluées avant l'addition `A` et la soustraction `S`, lesquelles sont donc effectuées en dernier lieu.

  Ainsi `2 - 2 * 2 renvoie -2` (et non 0), et `2 + 4 / 2 = 4.0` (et non 3.0).

- Si deux opérateurs ont la même priorité, l'évaluation est effectuée de gauche à droite. Ainsi dans l'expression `59 * 100 / 60`, la multiplication est effectuée en premier, et la machine doit donc ensuite effectuer `5900 / 60`, ce qui donne `98.0` (et non 59.0).

Racine carrée, fonctions trigonométriques...

Il est aussi possible d'effectuer des calculs plus complexes en utilisant par exemple des racines carrées, des fonctions trigonométriques...

import de la bibliothèque math

Pour utiliser ces fonctions mathématiques plus avancées, il est nécessaire d'ajouter une ligne au début de votre programme :

```
import math
```

Cette ligne permet d'importer (et donc d'utiliser) le module `math` (ce module contient toutes les fonctions mathématiques "classiques" que python ne possède pas de base dans la bibliothèque standard).

Voici quelques exemples :

- `math.pow(x,a)` permet de calculer x à la puissance a
- `math.cos(x)` permet de calculer le cosinus de l'angle x (l'angle x doit être en radian) (nous avons la même chose pour le sinus ou la tangente)
- `math.sqrt(x)` permet de calculer la racine carrée de x

Si vous avez besoin d'autres fonctions mathématiques, je vous invite à consulter la documentation de Python : https://docs.python.org/fr/3.7/library/math.html

Remarque : il existe de nombreuses bibliothèques (modules) qui apportent des collections d'instructions, comme `random` pour produire des nombres aléatoires ou encore `turtle` qui permet de reproduire les bases du langage éducatif Logo.

Exemples d'utilisation de la bibliothèque math

```
>>> import math
>>> a = 5
>>> b = 16
>>> c = 3.14 / 2
>>> math.pow(a,2)
25.0
>>> math.sqrt(b)
4.0
>>> math.sin(c)
0.9999996829318346
```

Remarque : en python, on utilise le point pour écrire la virgule dans un nombre.

D'autres types de variables

Les variables peuvent contenir autre chose que des nombres. Elles peuvent aussi contenir (entre autres) des suites de caractères, que l'on appelle "chaîne de caractères".

Une chaîne de caractères dans une variable

```
>>> maChaine = "Bonjour le monde !"
>>> print(maChaine)
Bonjour le monde !
>>> maChaine
'Bonjour le monde !'
```

Les variables peuvent donc contenir des types de données différents, pour l'instant nous en avons vu trois :

- le type "nombre entier" (integer en anglais, abréviation : `int`)
- le type "chaînes de caractères" (`string` en anglais)
- le type "nombre à virgule flottante" (`float` en anglais)

Il existe d'autres types de variables :

- le type booléen : `bool`
- les types composés : `tuple`, `list`, `dict`, ...

Nous aurons l'occasion de revenir sur ces derniers types de variable plus tard.

En Python les variables ont un type, mais le programmeur n'est pas obligé de préciser ce type. Il existe beaucoup de langage (C++, Java...) où l'utilisateur doit absolument définir le type d'une variable avant de pouvoir l'utiliser, faute de quoi cela entraînera une erreur.

Déterminer le type de variable

L'instruction `type` vous permet de connaître le type d'une variable, par exemple :

```
>>> a = "Salut !"
>>> b = 567
>>> c = 5.87
>>> type(a)
<class 'str'>
>>> type(b)
<class 'int'>
>>> type(c)
<class 'float'>
```

Comme vous pouvez le constater, la variable a est de type "chaînes de caractères" (`str` pour string), la variable b est de type "nombre entier" (`int` pour integer) et la variable c est de type "nombre à virgule flottante" (`float`).

Le signe + et les chaînes de caractères

L'utilisation du signe `+` ne se limite pas à l'addition. Il est aussi utilisé pour la `concaténation` .

D'après Wikipédia : « Le terme concaténation (substantif féminin), du latin cum («avec») et catena («chaîne, liaison»), désigne l'action de mettre bout à bout au moins deux chaînes. »

Comme vous avez pu le deviner en lisant la définition ci-dessus, la concaténation va concerner les chaînes de caractères.

Concaténation de chaînes

Observez le programme suivant :

```
>>> a = "Hello"
>>> b = "World"
>>> a + b
'HelloWorld'
```

On constate que les deux chaînes de caractères contenues dans les variables `a` et `b` ont été concaténées (mises bout à bout).

Addition de deux variables de type différent

Observez le programme suivant :

```
>>> a = "Hello"
>>> b = 3
>>> a + b
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

Comme vous pouvez le constatez votre programme renvoie une erreur, il n'est pas possible d'utiliser le signe + avec des variables de type différent (type string et type int par exemple).

Multiplication d'une chaîne de caractères par un entier

Il est par contre possible de multiplier une variable de type string par une variable de type int :

```
>>> a = "Hello"
>>> b = 3
>>> a * b
'HelloHelloHello'
```

Donner la parole à l'utilisateur

Ne trouvez-vous pas que pour l'instant cela manque un peu d'interactivité ? En effet, l'utilisateur de vos programmes est plutôt passif !

Heureusement l'instruction input va permettre aux utilisateurs d'entrer des données.

Utilisation de l'instruction input

Observez le programme suivant :

```
>>> age = input("Quel est votre âge ? ")
Quel est votre âge ? 20
>>> print("Vous avez ",age," ans.")
Vous avez  20  ans.
```

Remarque : La variable `age` va contenir la réponse entrée au clavier par l'utilisateur.

L'instruction `input` permet de récupérer les caractères saisis au clavier, on obtient alors une variable de type `string` :

```
>>> type(age)
<class 'str'>
```

Addition de deux entiers

Écrivons un programme qui demande à l'utilisateur d'entrer 2 nombres entiers et affiche le résultat de l'addition de ces 2 nombres. On utilisera les variables `a` et `b` pour les deux nombres entiers, et la variable `somme` pour le résultat :

```
a = input("Entrez le premier nombre : ")
b = input("Entrez le deuxième nombre : ")
somme = a + b
print(somme)

# on obtient
Entrez le premier nombre : 12
Entrez le deuxième nombre : 36
1236
```

Les valeurs obtenues grâce à l'instruction `input` sont forcément de type `string`, le signe `+` de la ligne `sommme = a + b` réalise une concaténation avec des chaînes de caractères.

Il faut alors convertir une variable de type `string` en une variable de type `int` en utilisant l'instruction `int()`, on parle de `transtypage` :

```
# pour transtypage d'une chaîne de caractères en un entier
>>> chaine = "6"
>>> valeur = int(chaine)
>>> chaine
'6'
>>> valeur
6
>>> type(chaine)
<class 'str'>
>>> type(valeur)
<class 'int'>

# plus rapide
>>> valeur = int(input("premiere valeur ? "))
```

Remarque : on peut aussi réaliser le transtypage d'un entier en une chaîne de caractères avec l'instruction `string()`.

Addition de deux entiers (avec transtypage de variables)

On peut alors corriger notre programme afin de réaliser correctement une addition comme demandé :

```
a = int(input("Entrez le premier nombre : "))
b = int(input("Entrez le deuxième nombre : "))
somme = a + b
print(somme)

# on obtient
Entrez le premier nombre : 12
Entrez le deuxième nombre : 36
48
```

Vrai ou faux ?

Tester l'égalité

Si quelqu'un vous dit que « 4 est égal à 5 », vous lui répondez quoi ? « C'est faux ».

Si maintenant la même personne vous dit que « 7 est égal à 7 », vous lui répondrez bien évidemment que « c'est vrai ».

En Python, le double égal `==` est l'opérateur d'égalité, qui peut être soit vrai (`True`) soit faux (`False`).

```
>>> 4 == 5
False
>>> 7 == 7
True
```

Tests d'égalité avec des variables

L'opérateur d'égalité sera souvent utilisé avec des variables.

```
>>> a = 4
>>> b = 5
>>> a == b
False
>>> a = 5
>>> a == b
True
```

ATTENTION : Il ne faut pas confondre l'opérateur d'égalité `==` avec l'opérateur d'affectation `=` qui est utilisé pour attribuer une valeur aux variables. La confusion entre ces 2 opérateurs est une erreur classique qui est parfois très difficile à détecter !

Tester la différence en python

Il est possible d'utiliser aussi l'opérateur « différent de » qui s'écrit `!=` .

```
>>> a = 4
>>> b = 5
>>> a != b
True
>>> a = 5
>>> a != b
False
```

Remarque : il existe d'autres opérateurs de comparaison :

- « strictement inférieur à » : `<`
- « strictement supérieur à » : `>`
- « inférieur ou égal à » : `<=`
- « supérieur ou égal à » : `>=`

N'hésitez pas à les tester dans le shell de Thonny !

Les booléens

A chaque fois ces opérateurs renverront True (vrai) ou False (faux), on appelera `booléen` une variable qui ne peut prendre que la valeur `True` ou `False` :

```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

Remarque : en programmation, une expression qui peut être vraie (`True`) ou fausse (`False`) s'appelle un prédicat, et les deux valeurs de vérité possibles d'un prédicat sont de type booléen (`bool`).

Les conditions

Nous allons maintenant étudier une structure fondamentale en programmation le « si ..... alors.....sinon.....».

L'idée de base est la suivante :

```
si condition:
    suite_instruction1
sinon:
    suite_instruction2
```

Si « condition » est True alors « suite_instruction1 » est exécuté et « suite_instruction2 » est ignoré. Sinon (sous-entendu que « condition » est False) « suite_instruction2 » est exécuté et « suite_instruction1 » est ignoré.

Notez l'utilisation d'un élément nouveau : le décalage de « suite_instruction1 » et de « suite_instruction2 ». Ce décalage est appelé `indentation`, il permet de rendre le code plus lisible. Dans le cas de l'utilisation d'un si/alors, l'indentation est obligatoire en Python.

Titi ou Toto ?

Observez les programmes ci-dessous (il s'agit de deux programmes identiques à part la valeur de la variable b qui est différente).

```
# cas 1
a = 4
b = 7
if a < b:
    print('Je suis toto.');
    print("Je n'aime pas titi.")
else:
    print('Je suis titi.')
    print("Je n'aime pas toto.")
print("En revanche, j'aime le Python.")

# on obtient
Je suis toto.
Je n'aime pas titi.
En revanche, j'aime le Python.
```

```
# cas 2
a = 4
b = 3
if a < b:
    print('Je suis toto.');
    print("Je n'aime pas titi.")
else:
    print('Je suis titi.')
    print("Je n'aime pas toto.")
print("En revanche, j'aime le Python.")

# on obtient
Je suis titi.
Je n'aime pas toto.
En revanche, j'aime le Python.
```

Suivant le cas, la condition a < b est vraie ou fausse, et le programme donne alors un résultat différent.

Des choix alternatifs avec elif

Imaginons maintenant un programme dans lequel il y trois conditions par exemple, le cas où on cherche à savoir si un nombre est positif, nul ou négatif. On peut alors utiliser l'instruction `elif` qui signifie `else if` et qui permet d'enchainer des cas alternatifs (on peut mettre à la suite plusieurs `elif`).

```
x = int(input("Entrez une valeur pour x : "))
if x < 0:
    print('negatif')
elif x == 0:
    print('zero')
else:
    print('positif')
```

Combiner les conditions

Un `if` peut contenir plusieurs conditions, nous aurons alors une structure de la forme :

```
si condition1 op_logique condition2:
    suite_instruction1
sinon:
    suite_instruction2
```

« op_logique » étant un opérateur logique.

Nous allons étudier 2 opérateurs logiques : le « ou » (noté en Python `or`) et le « et » (noté en Python `and`).

Remarque : Il existe aussi l'opérateur `not`, ces opérateurs ne s'appliquent qu'à des booléens.

Par exemple :

- (condition1 or condition2) est vrai si la condition1 ou la condition2 est vraie (voir les deux en même temps)
- (condition1 and condition2) est faux si la condition1 est vraie et la condition2 est fausse (il faut avoir la condition1 et la condition2 vraies en même temps).

Les résultats peuvent être regroupés dans ce que l'on appelle une table de vérité :

table de vérité pour le "or"

condition1	condition2	condition1 or condition2
vrai	vrai	vrai
vrai	faux	vrai
faux	vrai	vrai
faux	faux	faux

table de vérité pour le "and"

condition1	condition2	condition1 and condition2
vrai	vrai	vrai
vrai	faux	faux
faux	vrai	faux
faux	faux	faux

Le programme suivant compare les coordonnées (xa,ya) et (xb,yb) de deux points A et B et affiche les informations sur leur position respective :

```
if xa == xb and ya == yb:
 print("Les points sont confondus")
elif xa == xb or ya == yb:
 print("Les points sont alignés horizontalement ou verticalement")
else:
 print("Les points sont indépendants")
```

Les encadrements

Pour tester si une valeur est comprise dans un intervalle donné, on peut utiliser un programme comme ci-dessous.

```
valeur = int(input("Entrez une valeur : "))

if valeur >= 10 and valeur <= 20:
    print('la valeur entrée est dans l\'intervalle')

else:
    print('hors intervalle')
```

Mais une particularité des opérateurs de comparaison, à laquelle on pense trop rarement, est que ceux-ci peuvent être chaînés pour exprimer des encadrements.

```
valeur = int(input("Entrez une valeur : "))
if 10 <= valeur <= 20:
    print('la valeur entrée est dans l\'intervalle')
else:
    print('hors intervalle')
```

La boucle while

La notion de boucle est fondamentale en informatique. Une boucle permet d'exécuter plusieurs fois des instructions qui ne sont présentes qu'une seule fois dans le code.

La structure de la boucle `while` (aussi appelée boucle "tant que") est la suivante :

```
while condition:
    instruction1
    instruction2
suite programme
```

Tant que la condition reste vraie, les instructions à l'intérieur du bloc (partie indentée) seront exécutées.

Observez le programme suivant :

```
i = 0
while i < 10:
    print("i vaut : ",end="")

    print(i)

    i += 1        # équivalent à i = i + 1

print("C'est terminé.")

# on obtient
i vaut : 0
i vaut : 1
i vaut : 2
i vaut : 3
i vaut : 4
i vaut : 5
i vaut : 6
i vaut : 7
i vaut : 8
i vaut : 9
C'est terminé.
```

On constate que tant que la valeur de `i` reste strictement inférieure à 10, la partie indentée se répète (à chaque répétition la condition i < 10 est testée).

ATTENTION : Il faut bien veiller à ce que la condition testée ne soit pas indéfiniment vraie sinon le programme ne pourra pas sortir de la boucle : on parle de boucle infinie.

Remarque : le `end=""` dans le premier print empêche le retour à la ligne ce qui permet d'afficher la variable `i` juste à côté.

La boucle for

Il existe un autre type de boucle en Python : la boucle `for` qui permet de répéter un nombre de fois donné un bloc d'instructions.

La structure de la boucle `for` est la suivante :

```
for i in range(debut,fin):
    instruction1
    instruction2
suite programme
```

Nous aurons ici une boucle où la variable i prendra toutes les valeurs entières comprises entre debut et fin (debut inclus et fin exclu).

Remarque : La fonction range renvoie une séquence d'entiers

On peut écrire :

- range(fin) : cela renvoi une séquence d'entiers successifs allant de 0 à fin exclu
- range(debut, fin) : revoi une séquence d'entiers successifs allant de debut à fin exclu
- range(debut, fin, pas) : renvoi une séquence d'entiers allant de debut à fin exclu avec un pas donnée.

Une première boucle for

Observez le programme ci-dessous.

```
for i in range(0,10):
 print("i vaut : ",end="")
 print(i)
print("C'est terminé.")

# on obtient
i vaut : 0
i vaut : 1
i vaut : 2
i vaut : 3
i vaut : 4
i vaut : 5
i vaut : 6
i vaut : 7
i vaut : 8
i vaut : 9
C'est terminé.
```

Les boucles `for` et `while` sont interchangeables dans un programme, la boucle `while` étant souvent utilisée quand le programmeur ne connait pas à l'avance le nombre de répétitions que devra effectuer la boucle.

Imbrication de boucles

Des boucles peuvent contenir des boucles, on parle alors de boucles imbriquées.

Voici un exemple :

```
for x in range(3):
    for y in range(3):
        print(x,y)

# on obtient :
0 0
0 1
0 2
1 0
1 1
1 2
2 0
2 1
2 2
```

Indice de boucle

Dans les exemples précédents, les indices de boucles i, x et y ont été utilisés dans la boucle, mais parfois nous voulons juste itérer un certain nombre de fois une portion de code sans utiliser cet indice. Pour cela, la lettre représentant l'indice de boucle est remplacée par un underscore `_` :

```
for _ in range(5):
    print("Hello !")

# on obtient :
Hello !
Hello !
Hello !
Hello !
Hello !
```

La bibliothèque turtle

La bibliothèque turtle permet de reproduire les fonctionnalités du langage de programmation éducatif Logo.

Les instructions de ce langage font se déplacer une tortue munie d'un crayon à la surface d'une feuille virtuelle utilisant un repère avec abscisses et ordonnées comme en mathématique.

La tortue commence au centre de l'écran au point de coordonnées (0,0), les distances sont en pixels et les angles en degré.

Parmi les instructions, on trouve :

```
goto(x, y)       # aller au point de coordonnées (x, y)
forward(d)       # avancer de la distance d
backward(d)      # reculer de la distance d
left(alpha)      # pivoter à gauche d'un angle alpha (en °)
right(alpha)     # pivoter à droite d'un angle alpha (en °)
circle(r, alpha) # tracer un arc de cercle de rayon r sur un            angle
alpha
dot(r)           # tracer un point de rayon r
```

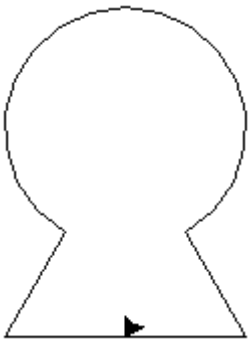
Avant de pouvoir utiliser la bibliothèque, il faut l'importer :

```
from turtle import *
```

Remarque : l'instruction précédente permet l'import de la totalité de la bibliothèque, ce qui n'est pas toujours le plus judicieux (nous reviendrons plus tard sur les différentes possibilités d'import).

Exemple d'utilisation :

```
from turtle import *
forward(60)
left(120)
forward(60)
right(90)
circle(60, 300)
right(90)
forward(60)
goto(0, 0)
```



D'autres instructions :

```
up()            # relever le crayon
down()          # repose le crayon
width(e)        # fixer à e l'épaisseur du trait
color(c)        # sélectionner la couleur c pour les traits
begin_fill()    # activer le mode remplissage
end_fill()      # désactiver le mode remplissage
fillcolor(c)    # sélectionner la couleur c pour le remplissage
reset()         # tout effacer et recommencer à zéro
speed(s)        # définir la vitesse de déplacement
title(t)        # donner un titre à la fenêtre de dessin
ht()            # n'affiche plus la tortue seulement le dessin
```

Remarques :

- les couleurs possibles sont : "black", "white", "grey", "blue", "red", "purple"...
- on peut aussi choisir la couleur avec color(r, v, b) où r, v, b sont des nombres entre 0 et 1 représentant les niveaux de rouge, vert et bleu
- les vitesses possibles sont : "slowest", "slow", "normal", "fast" et "fastest" ou un nombre entre 0 et 10

Retrouvez un aide mémoire sur la bibliothèque turtle :
https://perso.limsi.fr/pointal/_media/python🐢turtleref.pdf