Introduction au language Python

Jonas Grütter, Gymnase du Bugnon, le 27.05.18

Sources:

- Pensez en Python (https://allen-downey.developpez.com/livres/python/pensez-python/) par Allen B. Downey,
- The official home of the Python Programming Language (https://www.python.org/ (https://www.python.org/)),
- Python (https://fr.wikipedia.org/wiki/Python_(langage)) sur Wikipedia

Sources images:

- Guido Van Rossum: Twitter (https://twitter.com/gvanrossum/ (https://twitter.com/gvanrossum/))
- Thonny: Printscreen
- Mots-clès de Python: Pensez en Python (https://allen-downey.developpez.com/livres/python/pensez-python/)
- Instructions conditionnelles: quantrimang (https://quantrimang.com/lenh-if-ifelse-trong-python-141111 (https://quantrimang.com/lenh-if-ifelse-trong-python-141111))

Historique

Python est un langage de programmation orienté objet créé en 1990 par le programmeur néerlandais **Guido Van Rossum**. Etant fan de la série télévisée **Monthy Python's Flying Circus**, il décide de nommer son langage Python. Il est possible de percevoir beaucoup d'allusions à cette série dans les livres d'apprentissages de ce langage (Pensez Python par Allen Downey par exemple).

Le langage Python a été écrit dans le but d'être **simple à utiliser**. En effet, c'est un langage très facilement **lisible** et **abordable** qui fonctionne de manière très **logique**. Il y a par exemple bien moins de règles synthaxiques que dans d'autres langages moins abordables tels que le C, le Perl ou le Pascal.

Python est un **langage interprété** et non compilé. C'est à dire que chaque ligne du code source est analysée une par une et exécutée ensuite par un interpréteur. Les langages interprétés sont très intêressant grâce à leur facilité de mise en oeuvre et à la portabilité des programmes (il est possible de les lancer sur chaque plateforme ou fonctionne l'interpréteur).

Dans un langage compilé, le code écrit est compilé par un compilateur en code binaire facilement lisible par un ordinateur mais "illisible" par l'humain. Ensuite, le système d'exploitation lui-même va utilise le code binaire et les données d'entrée pour calculer les donnéees de sortie. Comme le langage compilé est directement exécuté sur l'ordinateur et qu'il n'a pas besoin d'interpréteur, il sera génerallement plus rapide qu'un langage interprété(pour le même programme). Ces deux types de langages présentent donc chacun des avantages et des inconvénients.

Par sa **performance** et sa **facilité d'accès**, Python a vite gagné en popularité et est devenu le **4ème langage informatique le plus utilisé** en Mai 2018 selon le TIOBE Index (Source : TIOBE Index (https://www.tiobe.com/tiobe-index/)). Ce langage est d'ailleurs fort recommandé pour faire ses **premiers pas en programmation** en raison de sa **facilité** et de son **utilité** dans la vie professionelle.

Execution de commandes dans la console

Quand Python est lancé dans un terminal Linux une invite >>> (**prompt** en anglais) est affiché sur l'écran. On peut y entrer une expression et avoir une réponse immédiate. Ici, dans ce notebook Jupyter, les entrées ne sont pas désignées par un prompt (>>>), mais par `In [i]` . La cellule qui contient la sortie s'appelle `Out[i]` . Les cellulles correspondent donc à la console. Voici un exemple d'une simple addition:

In [1]:

```
1 1+1
```

Out[1]:

2

Ou encore le programme classique d'afficher 'hello world':

In [2]:

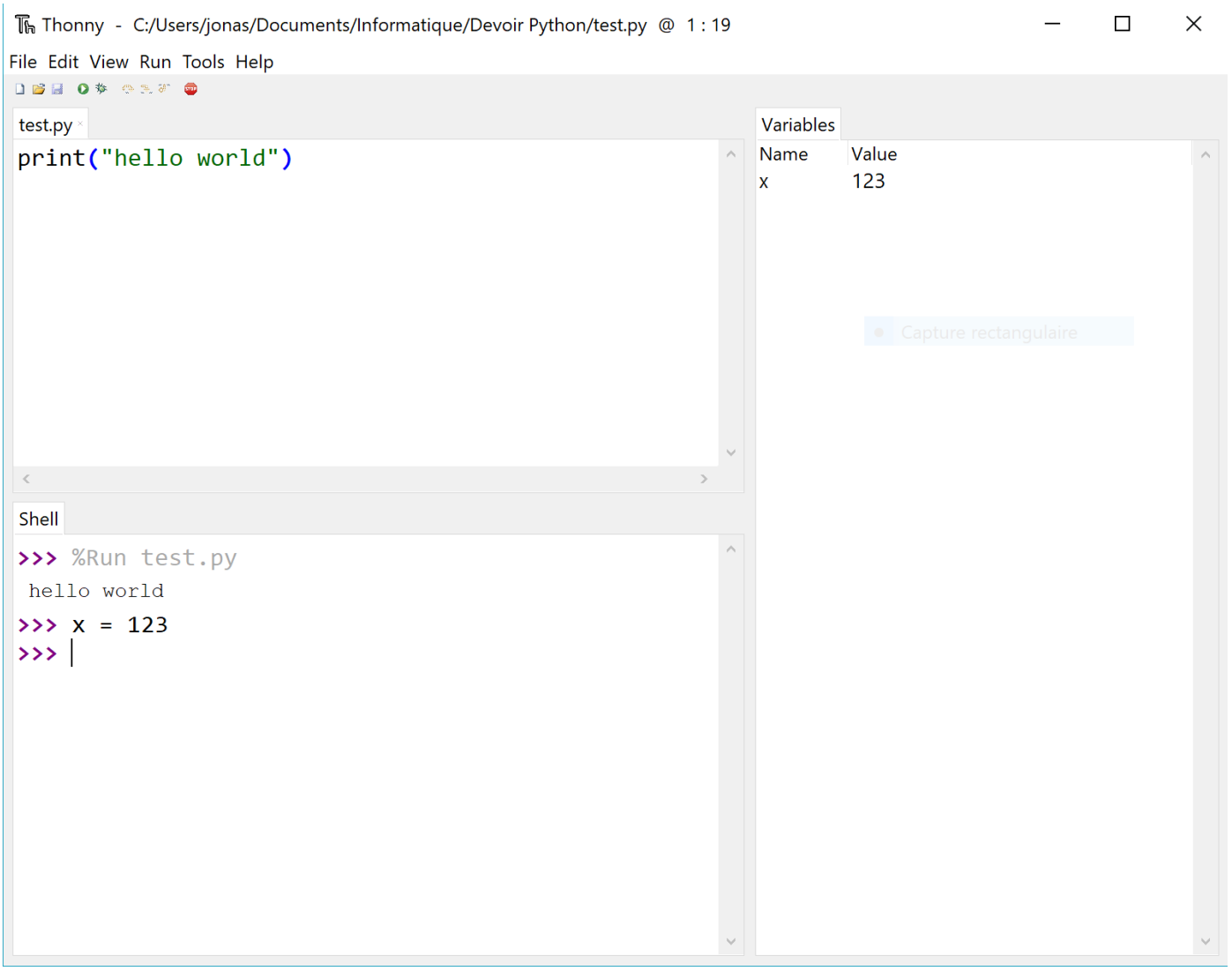
```
1 print('hello world')
```

```
hello world
```

Execution d'un script

Python est un langage qui est particulièrement utilisé comme **langage de script**. C'est à dire que les gens l'utilisent de façon à exécuter des ordres prédéfinis sauvegardés dans un fichier. Il est donc possible d'utiliser Python comme langage de script en écrivant dans le script une liste d'ordre

prédéfini(=un programme) et de laisser l'interpréteur interpréter le code. Il faut donc lancer le programme pour obtenir le résultat du code. Ces deux modes (consol et script) sont très facilement discernables dans l'interpéteur Thonny:



Dans Thonny, nous pouvons distinguer 3 régions.

- L'**éditeur de script** (en haut) dans lequel on écrit le code qui va être éxécuté lorsque le programme est lancé en appuyant sur le bouton d'execution (bouton vert).
- La **fenêtre de console** (en bas) ou les instructions sont éxécutées directements.
- La **fenêtre des variables** (à droite) ou les variables sont affichés.

Chaque valeur a un type

Une valeur est une untité d'information de base. Par exemple:

- nombre entier (123)
- nombre à virgule (1.23)
- chaîne de caractères ('123')
- valeur logique (True)

Pour vérifier à quel type appartient votre valeur, il est possible d'utiliser la fonction prédéfinie nommée `type()`, qui renvoie le type de la valeur.

Les **entiers** (en anglais integers) appartiennent au type **int**.

In [10]:

```
1 type(123)
```

Out[10]:

```
int
```

Les **nombres à virgule flottante** (en anglais floating-point numbers) appartiennent au type **float**.

In [3]:

```
1 type(1.23)
```

Out[3]:

```
float
```

Les **chaînes de caractères** (en anglais strings) appartiennent au type **str**.

In [14]:

```
1 type('spam and eggs')
```

Out[14]:

```
str
```

Attention! Des chiffres entre guillemets comme '42' sont du type str et non int. Même chose pour des nombres à virgule flottante entre guillemets.

In [1]:

```
1 type('42'), type('42.0')
```

Out[1]:

```
(str, str)
```

Comme nous le verrons plus tard, les **valeurs logiques** (True, False) appartiennent au type **bool**.

In [3]:

```
1 type(True)
```

Out[3]:

```
bool
```

Une variable peut stocker une valeur

Une variable est **une place dans la mémoire de l'ordinateur pour stocker une valeur**. Une des fonctionnalités les plus importantes d'un langage informatique est de pouvoir mémoriser et manipuler des valeurs. Une variable est un **nom qui fait référance à une valeur**.

Pour pouvoir créer une nouvelle variable, il faut utiliser l'**instruction d'affectaion** qui est le signe `=` et qui a la forme `var = valeur`. Voici la défintion de 4 variables:

In [7]:

```
a = 123
b = 1.23
c = '123'
d = True
print(a, b, c, d)
```

```
123 1.23 123 True
```

Ici, nous affectons une **chaine de caractères** à la variable `meal`. Il est important de voir que lorsque nous imprimons cette variable, c'est la chaîne de caractères affectée qui est imprimée. La première ligne est une instruction d'affectation qui donne un valeur à `meal` et la deuxième ligne est uine instruction print qui affiche la valeur de `meal`.

In [22]:

```
meal = 'spam and saussage'
print(meal)
```

```
spam and saussage
```

Attention à ne pas mettre la variable entre guillemets lors de son utilisation, car cela impliquerait que c'est une chaîne de caractéres et donc une nouvelle valeur du type str.

In [1]:

```
meal = 'spam and saussage'
print('meal', type("meal"))
```

```
meal <class 'str'>
```

Il est bien sûr aussi possible d'assigner des valeurs de **type int** et **float** à des variables.

In [1]:

```
number = 42
pi = 3.1415926
print(number, pi)
```

```
42 3.1415926
```

Noms de variables

Pour pouvoir se retrouver facilement dans un programme, il est important de donner un **nom significatif** aux variables afin de pouvoir exprimer leur **utilité**. Les noms de variables peuvent être aussi long que l'on veut, ils peuvent contenir des chiffres et des lettres et des lettres majuscules. Il est par contre coutume d'utiliser des noms de variables **courts** et **sans majuscule** afin de se simplifier la vie.

Il y a par contre certaines règles à adopter pour les noms de variable:

Le nom d'une variable ne peut **pas commencer par un chiffre**.

In [28]:

```
1 007agent = 'James Bond'
```

```
  File "<ipython-input-28-9b050cc100d0>", line 1
    007agent = 'James Bond'
      ^
SyntaxError: invalid token
```

Certains **caractères** comme l'arobase ne sont **pas autorisés**

In [29]:

```
1 food@home = 'spam'
```

```
  File "<ipython-input-29-c2dfb8674d34>", line 1
    food@home = 'spam'
        ^
SyntaxError: invalid syntax
```

Une variable ne peut pas être nommée comme un des **mots-clés** de Python. L'interpréteur utilise certains mots pour reconnaître la structure du programme. Ces mots-clés sont les suivants:

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

Pour pouvoir les **reconnaître** sans avoir besoin de les apprendre, ils s'affichent d'une **autre couleur** que le code standard.

In [30]:

```
1 finally = 'We still eat spam'
```

```
  File "<ipython-input-30-52d83cb458e6>", line 1
    finally = 'We still eat spam'
          ^
SyntaxError: invalid syntax
```

Pour donner un **nom composé à une variable**, il est d'usage d'utiliser le caractèrre de soulignement `_` .

In [62]:

```
1 dish_of_the_day = 'spam and tomato'
```

Expressions

Une expression est une combinaison de valeurs, de variables, d'opérateurs et de fonctions qui est évaluée en suivant les règles de priorité et d'associativité du langage de programmation pour retourner une nouvelle valeur. (Source : Expressions (https://fr.wikipedia.org/wiki/Expression_(informatique)) sur Wikipédia)

Opérateurs arithmétiques

Il est nécessaire de parler des **opérateurs arithmétiques**. Python est un langage qui fournit des opérateurs arithmétiques **permettant de faire des calculs**. Ce sont des **symboles** spéciaux qui représentent des calculs.

L'opérateur `+` représente **l'addition**

In [35]:

```
1 42 + 42
```

Out[35]:

84

L'opérateur `-` représente la **soustraction**

In [59]:

```
1 50 - 8
```

Out[59]:

42

L'opérateur `*` représente la multiplication

In [37]:

```
1 21 * 2
```

Out[37]:

42

L'opérateur `/` représente la **division** et renvoie une valeur de **type float** afin d'exprimer le **chiffre à virgule**

In [14]:

```
1 x = 43 / 3
2 print(x, type(x))
```

```
14.333333333333334 <class 'float'>
```

L'opérateur `//` représente la **division entière**, le résultat sera de **type int**. Il est à noter que l'arrondie se fait toujours vers le bas.

In [15]:

```
1 x = 43 // 3
2 print(x, type(x))
```

```
14 <class 'int'>
```

L'opérateur `**` représente l'**exponentiation**.

In [61]:

```
1 2**3
```

Out[61]:

```
8
```

L'opérateur **%** représente **l'opérateur modulo**. Il effectue la **division** et renvoie le **reste**

In [81]:

```
1 42 % 5
```

Out[81]:

```
2
```

Le langage de programmation Python **respecte les règles de priorité** dans les opérations.

In [4]:

```
1 5 + 6 / 2, (5+6)/2
```

Out[4]:

```
(8.0, 5.5)
```

L'opérateur `+` dans le contexte d'une chaîne de caractères est définit comme la **concaténation**.

In [41]:

```
1 "Monty" + " " + "Python"
```

Out[41]:

```
'Monty Python'
```

L'opérateur `*` dans le contexte d'une chaîne de caractères est définit comme la **répétition**.

In [44]:

```
1 "spam" * 10
```

Out[44]:

```
'spamspamspamspamspamspamspamspamspamspam'
```

Expessions boléennes

Une **expression boléenne** est une expression qui est soit **vraie**, soit **fausse**. Si l'expression est **vraie**, elle renvoie **True** (vrai). Si elle est **fausse**, elle renvoie **False** (faux). Les opérateurs utilisés ici sont dit **relationnels**.

L'opérateur `==` est **vrai** si les valeurs sont **égales**.

In [2]:

```
1 42 == 42, "spam" == "tasty meal"
```

Out[2]:

```
(True, False)
```

L'opérateur `!=` est **vrai** si les valeurs ne sont **pas égales**.

In [69]:

```
1 42 != 42
```

Out[69]:

```
False
```

L'opérateur `<` est **vrai** si la valeur de gauche est **strictement inférieur** à celle de droite.
L'opérateur `>` test l'inverse.

In [24]:

```
1 12 < 13,  12 > 13
```

Out[24]:

```
(True, False)
```

L'opérateur `<=` est **vrai** si la valeur de gauche est **inférieure ou égale** à celle de droite.
L'opérateur `>=` test l'inverse.

In [25]:

```
1 12 <= 12, 12 >= 13
```

Out[25]:

```
(True, False)
```

True et **False** sont des **valeurs** appartenant au **type bool**. Ce ne sont pas des chaînes de caractères.

In [76]:

```
1 type(True)
```

Out[76]:

```
bool
```

Opérateurs logiques

Il y a 3 opérateurs logiques dans Python. Ces opérateurs sont `and` , `or` et `not` .

Le mot `and` se traduit par le mot **et** en français. Une expression contenant l'opérateur logique `and` ne sera vraie que si toutes les conditions sont vraies.

In [5]:

```
1 12 < 42 and 20 < 42, 12 < 42 and 43 < 42
```

Out[5]:

```
(True, False)
```

L'opérateur `or` se traduit par le mot **ou**. Une expression contenant cet opérateur logique sera vraie si l'une des condition est vraie.

In [88]:

```
1 12 == 13 or 12 < 13
```

Out[88]:

```
True
```

L'opérateur `not` se traduit par le mot **non**. C'est une opérateur qui nie une expression boléenne. Une expression contenant `not` n'est vraie que si elle est fausse lorsqu'elle ne contient pas le `not` .

In [2]:

```
1 not 12 == 12
```

Out[2]:

```
False
```

Application des expressions avec des variables

Les expressions montrées précedemment sont d'une très grande utilité lorsque l'on les utilise avec des variables. En effet, cela permet de gagner du temps.

Ici, pour le même calcul, il suffira juste de changer la valeur affectée à la variable pour pouvoir obtenir le résultat de ce même calcul avec un autre nombre.

In [90]:

```
x = 10
calcul = x**2*50/100
print(calcul)
```

```
50.0
```

ou encore

In [102]:

```
minutes = 142
heures = minutes //  60
reste = minutes - heures * 60
print(heures, "heures", reste, "minutes")
```

```
2 heures 22 minutes
```

Structure de contrôle

En programmation informatique, une structure de contrôle est une instruction particulière qui peut dévier le flot de contrôle du programme.

Source : Structure de contrôle (https://fr.wikipedia.org/wiki/Structure_de_contrôle) sur Wikipédia

Instruction conditionnelle

Source : Instruction conditionnelle (https://fr.wikipedia.org/wiki/Instruction_conditionnelle_(programmation)) sur Wikipédia

Le **flux d'exécution** est **l'ordre dans lequel les instructions sont exécutées**. L'exécution commence toujours par la première instruction du programme et les autres instructions sont exécutées une par une dans l'ordre de haut en bas.

Il est important de savoir qu'un **appel de fonction** est comme un **détour dans le flux d'exécution**. Au lieu de passer à l'instruction suivante, le flux se dirige dans le corps de la fonction dans laquelle il exécute les instructions. Après cela, il reprend son 'chemin' là ou il s'était interrompu plus tôt.

Pour réussir à **mieux comprendre un programme**, il est souvent préférable de le **lire dans l'ordre de son flux d'exécution** et non dans l'ordre dans lequel il a été écrit.

Il est important de savoir que lorsque l'on fait tourner un programme, **il n'y pas forcément toutes les instructions qui sont exécutées**. Certaines ne sont exécutées que **sous certaines conditions**. Il va donc y avoir dans le programme des **instructions conditionnelles** qui vont 'tester' la valeur. La direction que prendra le flux d'exécution dépend du résultat du test. Il faut donc vérifier des conditions et modifier le comportement du programme en conséquence.

Nous allons voir ici 3 mots-clés pour contrôler le flux conditionnel: `if` , `else` , et `elif` .

If, else, elif

La **structure** des exécutions condititionelles sont dites **composées**. Cela veut dire qu'il y a un **en-tête**, suivi d'un **corps d'identité**. Il n'y pas de limite sur le nombre d'instruction pouvant apparaître dans le corps d'identité, mais il est obligatoire d'en mettre au moins une. Il est toutefois possible d'utiliser l'instruction `pass` qui dit qu'il ne faut rien faire.

L'instruction `if` est la forme la plus **simple des instructions conditionnelles**. Le mot `if` est traduit par le mot *si* en français.

Après l'instruction conditionelle, il faut placer une **expression boléenne** qui est appelée condition. Si elle est vraie, le flux d'exécution se dirigera de manière à exécuter l'instruction donnée après le `if` . Sinon, l'instruction corresponddant à l'altérnative `else` est exécutée.

In [16]:

```
hungry = True
if hungry:
    print('eggs')
else:
    print('spam')
```

```
eggs
```

Mais si la condition n'est pas vrai, l'instruction correspondant à l'alternative est executée.

In [18]:

```
hungry = False
if hungry:
    print('eggs')
else:
    print('spam')
```

```
spam
```

L'instruction `else` est l'**instruction alternative** du `if` . Le `else` peut être traduit en français par sinon. Cela veut dire que lorsqu'on place une instruction `if` et que l'expression boléenne qui la suit est fausse, alors le programme exécutera le corps d'identité suivant le `else` .

Pour être plus clair, prenons un autre exemple:

In [6]:

```
x = 4
if x % 3 == 0:
    print("x est divisible par trois")
else:
    print("x n'est pas divisble par trois")
```

```
x n'est pas divisble par trois
```

Dans cet exemple, l'expression boléenne suivant le `if` test si le nombre x est divisible par 3. Si le reste de la division de x par 3 est égal à 0, le programme exécute les instructions suivant le if et non celles suivant le `else` . En revanche, si le nombre n'est pas divisible par 3, la deuxième série d'instructions sera exécutée.

Il est important de savoir que comme l'expression boléenne est soit vraie, soit fausse, **une seule des branches alternative sera exécutée**.

L'instruction `elif` est une abréviation de `else if` . Elle est utile quand il y a **plus de deux possibilités** et que nous avons besoin de **plus de deux branches**.

Le code ci-dessous montre bien que l'on peut avoir besoin de plus de deux branches, mais il n'est pas écrit de manière optimale:

In [7]:

```
x = 6
y = 4
if x < y:
    print("x est plus petit que y")
else:
    if x > y:
        print("x est plus grand que y")
    else:
        if x == y:
            print("x est égal à y")
```

```
x est plus grand que y
```

Il est possible de voir par la que les structures conditionnelles peuvent être **imbriquées** les unes dans les autres. La première branche (ligne 4-5) contient une simple instruction. La deuxième (ligne 7-12), qui contient elle-même une instruction if et else, a deux branches elle aussi (ligne 8-12).

Dans ce cas la, ces deux branches (ligne 8-12) contiennent de simples instructions, mais elles auraient pu elles ausssi contenir encore des instructions conditionelles et ainsi de suite. **Les conditions imbriquées rendent le code très difficile à lire**. L'instruction `elif` et les connecteurs logiques permettent de les éviter et donc écrire le code de manière plus claire.

Réécrivons le code ci-dessus de manière plus claire grâce à l'instruction `elif` :

In [8]:

```
x = 6
y = 4
if x < y:
    print("x est plus petit que y")
elif x > y:
    print("x est plus grand que y")
else:
    print("x est égal à y")
```

```
x est plus grand que y
```

Le shéma ci-dessous correspond au code ci-dessus. Lôbserver attendivement permet de mieux comprendre les instructions conditionelles.

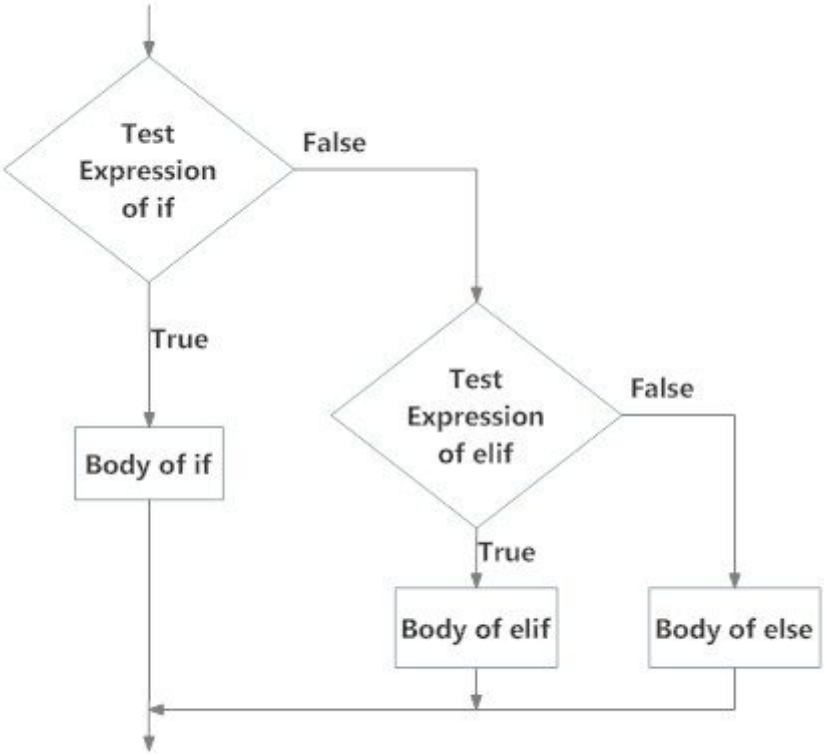


Fig: Operation of if...elif...else statement

Dans ce cas là, chaque condition est vérifiée dans l'ordre. Dès qu'une condition est vraie, la branche correspondante est exécutée. Même si plusieurs conditions sont vraies, seule la branche de la première condition vraie est exécutée. Il est encore important de noter qu'il n'y a pas de limite du nombre d'instruction `elif` à mettre et la clause `else` à la fin n'est pas obligatoire.

La boucle

En Python, l'instruction `for` sert à itérer sur les éléments d'une séquence (liste, chaîne de caractère,etc.) dans l'ordre dans lequel ils apparaissent dans la séquence (Source : [L'instruction for (https://docs.python.org/fr/3/reference/compound_stmts.html#the-for-statement)](https://docs.python.org/fr/3/reference/compound_stmts.html#the-for-statement)).

L'utilisation d'une instruction 'for element in sequence' suivi d'une instruction correspondante, peut se traduire comme ceci: **Pour tous éléments de la séquence, il faut répéter la même instruction donnée**. On parle donc de **boucle** car il faut répéter la même instruction pour chaque éléments.

Prenons par exemple le menu quotidien et original des Monty Python. Dans cet exemple, je demande que, pour tous les éléments de la liste nommée menu, il faut imprimer le nom de l'élément ainsi que le nombre de lettre de l'élément grâce à la fonction `len()`.

In [11]:

```
menu = ['spam', 'eggs', 'saussage', 'ham']
for item in menu:
    print(item, len(item))
```

```
spam 4
eggs 4
saussage 8
ham 3
```

Il est aussi possible d'utiliser l'instruction `for` pour une chaine de caractères. Dans cet exemple, je demande que chaque caractère soit imprimé un par un.

In [21]:

```
dish = 'spam'
for car in dish:
    print(car)
```

```
s
p
a
m
```

La fonction `range(n)` génère une liste contenant une simple progression arithmétique contenant n éléments (de 0 à n-1). Lorsque l'on l'utilise avec l'instruction `for`, cela donne la possibilité de répéter une action un nombre de fois donné. Ici, je demande d'imprimer le mot spam 4 fois:

In [29]:

```
for i in range (4):
    print("spam")
```

```
spam
spam
spam
spam
```

Lorsque l'on combine les fonctions `range()` et `len()`, il est aussi possible d'intéragir avec une liste par exemple:

In [2]:

```
menu = ['eggs', 'saussage', 'ham', 'spam']
n = len(menu) # len(menu) = nombre d'élément dans menu
for i in range(n):
    print(i, menu[i], ", It's tasty")
```

```
0 eggs , It's tasty
1 saussage , It's tasty
2 ham , It's tasty
3 spam , It's tasty
```

Dans ce dernier exemple, pour chaque élément de la liste, j'ai imprimé le numéro qu'il porte dans la liste, le nom de l'élément et j'ai encore affirmé que l'élément était bon à manger.