

Une introduction à Python

Ivan Noyer

Lycée Thiers

Sommaire

1 Un peu de logique

2 Présentation de Python

- Historique
- Structure
- Types
- Variables et mots clés
 - Variables
 - Opérateurs
- Expressions et instructions
 - Définitions

- Séquences d'instructions
- Expressions conditionnelles
- Boucles
- Fonctions et procédures
- Importations de modules

3 Compléments

- Variable mutable VS immuable
- Listes
- Objets `range`
- Affectation VS expression

Mes pages

<https://nussbaumcpge.be/>

Tous les cours, les TD, les TP et les corrigés des devoirs.

- Informatique pour tous en classes préparatoires aux grande écoles (Eyrolles)

- Informatique pour tous en classes préparatoires aux grande écoles (Eyrolles)
- Wikipedia

- Informatique pour tous en classes préparatoires aux grande écoles (Eyrolles)
- Wikipedia
- OpenClassRoom

Tables de vérité

Les 5 opérateurs de la logique classique

Table du ET		
A	B	$A \wedge B$
V	V	V
V	F	F
F	V	F
F	F	F

Table du OU		
A	B	$A \vee B$
V	V	V
V	F	V
F	V	V
F	F	F

Table du XOR OU Exclusif		
A	B	$A \oplus B$
V	V	F
V	F	V
F	V	V
F	F	F

Table du NOT	
A	$\neg A$
V	F
F	V

Table d'implication		
A	B	$A \Rightarrow B$
V	V	V
V	F	F
F	V	V
F	F	V

Les opérateurs logiques en Python

- $\wedge$: `and`

Les opérateurs logiques en Python

- $\wedge$: `and`
- $\vee$: `or`

Les opérateurs logiques en Python

- $\wedge$: `and`
- $\vee$: `or`
- $A \implies B$: `if A: B`

Guido van Rossum

Benevolent Dictator for Life (BDFL) : surnom donné à une personne respectée de la communauté de développement open source qui définit des orientations générales d'un projet donné.

Littéralement *Bienveillant dictateur à vie*. Ce nom est un jeu de mots entre dictateur bienveillant et président à vie.

FIGURE – Guido van Rossum, dictateur bienveillant à vie du projet Python



Historique

- À la fin des années 1980, le programmeur Guido van Rossum participe au développement du langage de programmation ABC au Centrum voor Wiskunde en Informatica (CWI) d'Amsterdam, aux Pays-Bas.

Historique

- À la fin des années 1980, le programmeur Guido van Rossum participe au développement du langage de programmation ABC au Centrum voor Wiskunde en Informatica (CWI) d'Amsterdam, aux Pays-Bas.
- Il estime alors qu'un langage de script inspiré d'ABC pourrait être intéressant comme interpréteur de commandes pour Amoeba (un système d'exploitation).

Historique

- À la fin des années 1980, le programmeur Guido van Rossum participe au développement du langage de programmation ABC au Centrum voor Wiskunde en Informatica (CWI) d'Amsterdam, aux Pays-Bas.
- Il estime alors qu'un langage de script inspiré d'ABC pourrait être intéressant comme interpréteur de commandes pour Amoeba (un système d'exploitation).
- En 1989, profitant d'une semaine de vacances durant les fêtes de Noël, il utilise son ordinateur personnel pour écrire la première version du langage. Fan de la série télévisée des Monty Python, il décide de baptiser ce projet Python.

Historique

- À la fin des années 1980, le programmeur Guido van Rossum participe au développement du langage de programmation ABC au Centrum voor Wiskunde en Informatica (CWI) d'Amsterdam, aux Pays-Bas.
- Il estime alors qu'un langage de script inspiré d'ABC pourrait être intéressant comme interpréteur de commandes pour Amoeba (un système d'exploitation).
- En 1989, profitant d'une semaine de vacances durant les fêtes de Noël, il utilise son ordinateur personnel pour écrire la première version du langage. Fan de la série télévisée des Monty Python, il décide de baptiser ce projet Python.
- Février 1991 : première version publique 0.9.07, postée sur le forum Usenet `alt.sources`.

Historique

- À la fin des années 1980, le programmeur Guido van Rossum participe au développement du langage de programmation ABC au Centrum voor Wiskunde en Informatica (CWI) d'Amsterdam, aux Pays-Bas.
- Il estime alors qu'un langage de script inspiré d'ABC pourrait être intéressant comme interpréteur de commandes pour Amoeba (un système d'exploitation).
- En 1989, profitant d'une semaine de vacances durant les fêtes de Noël, il utilise son ordinateur personnel pour écrire la première version du langage. Fan de la série télévisée des Monty Python, il décide de baptiser ce projet Python.
- Février 1991 : première version publique 0.9.07, postée sur le forum Usenet `alt.sources`.
- Python Software Foundation (PSF) : une association sans but lucratif fondée en 2001, détient la licence de Python depuis la version 2.1.

Historique

- À la fin des années 1980, le programmeur Guido van Rossum participe au développement du langage de programmation ABC au Centrum voor Wiskunde en Informatica (CWI) d'Amsterdam, aux Pays-Bas.
- Il estime alors qu'un langage de script inspiré d'ABC pourrait être intéressant comme interpréteur de commandes pour Amoeba (un système d'exploitation).
- En 1989, profitant d'une semaine de vacances durant les fêtes de Noël, il utilise son ordinateur personnel pour écrire la première version du langage. Fan de la série télévisée des Monty Python, il décide de baptiser ce projet Python.
- Février 1991 : première version publique 0.9.07, postée sur le forum Usenet `alt.sources`.
- Python Software Foundation (PSF) : une association sans but lucratif fondée en 2001, détient la licence de Python depuis la version 2.1.
- Python a été amélioré sur le principe de la compatibilité ascendante. Principe cassé lors du passage de la version 2.7 à 3.0.

Utilisation

- Python peut s'utiliser dans de nombreux contextes et s'adapter à tout type d'utilisation grâce à des bibliothèques spécialisées à chaque traitement. **Il est particulièrement utilisé comme langage de script pour automatiser des tâches simples mais fastidieuses.**
Pas prévu normalement pour l'IA d'un robot explorateur de comètes.

Utilisation

- Python peut s'utiliser dans de nombreux contextes et s'adapter à tout type d'utilisation grâce à des bibliothèques spécialisées à chaque traitement. **Il est particulièrement utilisé comme langage de script pour automatiser des tâches simples mais fastidieuses.**
Pas prévu normalement pour l'IA d'un robot explorateur de comètes.
- Plusieurs entreprises mentionnent sur leur site officiel qu'elles utilisent Python : Google (Guido van Rossum y a travaillé entre 2005 et 2012) ; la NASA, CCP Games (les créateurs du jeu vidéo EVE Online) etc. . .

Utilisation

- Python peut s'utiliser dans de nombreux contextes et s'adapter à tout type d'utilisation grâce à des bibliothèques spécialisées à chaque traitement. **Il est particulièrement utilisé comme langage de script pour automatiser des tâches simples mais fastidieuses.**
Pas prévu normalement pour l'IA d'un robot explorateur de comètes.
- Plusieurs entreprises mentionnent sur leur site officiel qu'elles utilisent Python : Google (Guido van Rossum y a travaillé entre 2005 et 2012) ; la NASA, CCP Games (les créateurs du jeu vidéo EVE Online) etc. . .
- Python est aussi le langage de commande d'un grand nombre de logiciels libres : FreeCAD (CAO 3D), Blender (modélisation 3D), XBMC (lecteur multimédia), Libre Office (suite bureautique) etc. . .

Pédagogie

- Une syntaxe épurée,

Pédagogie

- Une syntaxe épurée,
- Des environnements de développement très attrayants.

Pédagogie

- Une syntaxe épurée,
- Des environnements de développement très attrayants.
- MAIS des critères de passage de paramètres (par valeur ? par référence ?) déroutants pour le débutant.

Syntaxe

Le langage Python :

- Est conçu pour être un langage lisible, visuellement épuré.

Syntaxe

Le langage Python :

- Est conçu pour être un langage lisible, visuellement épuré.
- Utilise des mots anglais fréquemment, là où d'autres langages utilisent de la ponctuation.

Syntaxe

Le langage Python :

- Est conçu pour être un langage lisible, visuellement épuré.
- Utilise des mots anglais fréquemment, là où d'autres langages utilisent de la ponctuation.
- Possède également moins de constructions syntaxiques que de nombreux langages structurés tels que C, Perl, ou Pascal.

Syntaxe

Le langage Python :

- Est conçu pour être un langage lisible, visuellement épuré.
- Utilise des mots anglais fréquemment, là où d'autres langages utilisent de la ponctuation.
- Possède également moins de constructions syntaxiques que de nombreux langages structurés tels que C, Perl, ou Pascal.
- Commentaires : indiqués par le caractère hashtag #.

Syntaxe

Le langage Python :

- Est conçu pour être un langage lisible, visuellement épuré.
- Utilise des mots anglais fréquemment, là où d'autres langages utilisent de la ponctuation.
- Possède également moins de constructions syntaxiques que de nombreux langages structurés tels que C, Perl, ou Pascal.
- Commentaires : indiqués par le caractère hashtag #.
- Blocs d'instructions : identifiés par l'indentation. En C et C++ : accolades, en Pascal ou $\text{\LaTeX}$: `begin...end`.

Syntaxe

Le langage Python :

- Est conçu pour être un langage lisible, visuellement épuré.
- Utilise des mots anglais fréquemment, là où d'autres langages utilisent de la ponctuation.
- Possède également moins de constructions syntaxiques que de nombreux langages structurés tels que C, Perl, ou Pascal.
- Commentaires : indiqués par le caractère hashtag #.
- Blocs d'instructions : identifiés par l'indentation. En C et C++ : accolades, en Pascal ou $\text{\LaTeX}$: `begin...end`.
- Augmentation de l'indentation : début d'un bloc.

Syntaxe

Le langage Python :

- Est conçu pour être un langage lisible, visuellement épuré.
- Utilise des mots anglais fréquemment, là où d'autres langages utilisent de la ponctuation.
- Possède également moins de constructions syntaxiques que de nombreux langages structurés tels que C, Perl, ou Pascal.
- Commentaires : indiqués par le caractère hashtag #.
- Blocs d'instructions : identifiés par l'indentation. En C et C++ : accolades, en Pascal ou $\text{\LaTeX}$: `begin...end`.
- Augmentation de l'indentation : début d'un bloc.
- Réduction de l'indentation : fin du bloc courant.

Syntaxe

Le langage Python :

- Est conçu pour être un langage lisible, visuellement épuré.
- Utilise des mots anglais fréquemment, là où d'autres langages utilisent de la ponctuation.
- Possède également moins de constructions syntaxiques que de nombreux langages structurés tels que C, Perl, ou Pascal.
- Commentaires : indiqués par le caractère hashtag #.
- Blocs d'instructions : identifiés par l'indentation. En C et C++ : accolades, en Pascal ou $\text{\LaTeX}$: `begin...end`.
- Augmentation de l'indentation : début d'un bloc.
- Réduction de l'indentation : fin du bloc courant.
- Parenthèses facultatives dans les structures de contrôle.

Syntaxe

FIGURE – Comparaison de code

Fonction factorielle en C	Fonction factorielle en Python
<pre>/* Fonction factorielle en C */ int factorielle(int x) { if (x < 2) { return 1; } else { return x * factorielle(x-1); } }</pre>	<pre># Fonction factorielle en Python def factorielle(x): if x < 2: return 1 else: return x * factorielle(x-1)</pre>

Remarquer qu'en C, au moins, le type des données d'entrée et de sortie est précisé !

Quelques types

Les principaux types que nous verrons cette année :

- les types numériques : entiers et flottants (pour les calculs numériques),

Quelques types

Les principaux types que nous verrons cette année :

- les types numériques : entiers et flottants (pour les calculs numériques),
- le type booléen (pour les tests),

Quelques types

Les principaux types que nous verrons cette année :

- les types numériques : entiers et flottants (pour les calculs numériques),
- le type booléen (pour les tests),
- les types énumérables : tuples, chaînes de caractères et listes (pour les énumérations)

Quelques types

Les principaux types que nous verrons cette année :

- les types numériques : entiers et flottants (pour les calculs numériques),
- le type booléen (pour les tests),
- les types énumérables : tuples, chaînes de caractères et listes (pour les énumérations)
- les dictionnaires

Quelques types

Les principaux types que nous verrons cette année :

- les types numériques : entiers et flottants (pour les calculs numériques),
- le type booléen (pour les tests),
- les types énumérables : tuples, chaînes de caractères et listes (pour les énumérations)
- les dictionnaires
- D'autres types existent mais que nous utiliserons moins, comme les ensembles et les complexes.

Quelques types

Les principaux types que nous verrons cette année :

- les types numériques : entiers et flottants (pour les calculs numériques),
- le type booléen (pour les tests),
- les types énumérables : tuples, chaînes de caractères et listes (pour les énumérations)
- les dictionnaires
- D'autres types existent mais que nous utiliserons moins, comme les ensembles et les complexes.
- On peut créer des *classes* qui introduisent alors de nouveaux types.

Les noms des types Python

- `int` et `float`

Les noms des types Python

- `int` et `float`
- `bool`

Les noms des types Python

- `int` et `float`
- `bool`
- `tuple`, `str` et `list`

Les noms des types Python

- `int` et `float`
- `bool`
- `tuple`, `str` et `list`
- `dict` (et `instance` pour les objets d'une classe personnelle)

Les noms des types Python

- `int` et `float`
- `bool`
- `tuple`, `str` et `list`
- `dict` (et `instance` pour les objets d'une classe personnelle)
- `set` et `complex`

Type vs Ensemble — définitions en quelques mots

- **Ensemble** : collection d'objets définie *par une propriété* (vision *extensionnelle*).

Notation : $x \in S$ signifie « x appartient à S ».

- **Type** : classification de *termes bien formés* avec des *règles de construction/usage* (vision plutôt *intensionnelle*).

Notation : $t : A$ est un *jugement de typage*.

Idée clé : l'ensemble dit *qui appartient*, le type dit *comment on construit et manipule* des termes sûrs.

Contrastes essentiels

Ensemble ($\in$)

Vérité « $x \in S$ » à démontrer (proposition).

Défini par propriété : $\{x \mid P(x)\}$.

Égalité surtout extensionnelle (mêmes éléments).

Maths pures (théorie des ensembles).

Tolère « objets » hétérogènes.

Type ($:$)

Jugement syntaxique « $t : A$ » vérifié par des règles.

Défini par constructeurs/éliminateurs (produits, sommes, dépendants).

Égalité guidée par calcul/convertibilité.

Programmation/preuves *via* Curry–Howard (types $\simeq$ propositions).

Empêche les expressions sans sens (ex. : $1 + \text{True}$ interdit).

Mnémono : $\in$ affirme une appartenance ; $:$ atteste d'une construction bien typée.

Mots clés du langage

```
1 import keyword
2 print(keyword.kwlist)
```

On obtient :

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

Mots clés du langage

- Les mots clés sont accessibles depuis le *module* **keyword**.

Mots clés du langage

- Les mots clés sont accessibles depuis le *module* **keyword**.
- On ne peut pas créer de *fonction* ou de *variable* dont le nom est un mot clé.

Mots clés du langage

- Les mots clés sont accessibles depuis le *module* **keyword**.
- On ne peut pas créer de *fonction* ou de *variable* dont le nom est un mot clé.
- Les versions 3.* ne reconnaissent plus `print` et `exec` comme mots clés mais comme des *méthodes* du module **builtins**.

Noms de variables

- Le nom d'une variable est composé de lettre **a** à **z**, **A** à **Z**, et de chiffres **0** à **9**, mais ne doit pas commencer par un chiffre.

Noms de variables

- Le nom d'une variable est composé de lettre **a** à **z**, **A** à **Z**, et de chiffres **0** à **9**, mais ne doit pas commencer par un chiffre.
- Seules les lettres ordinaires sont autorisées. Les lettres accentuées, les cédilles, les espaces, les caractères spéciaux tels que **#**, **@** etc. sont interdits, à l'exception du caractère **_** (underscore). Le tiret **-** est bien sûr interdit puisqu'il correspond aussi à la soustraction.

Noms de variables

- Le nom d'une variable est composé de lettre **a** à **z**, **A** à **Z**, et de chiffres **0** à **9**, mais ne doit pas commencer par un chiffre.
- Seules les lettres ordinaires sont autorisées. Les lettres accentuées, les cédilles, les espaces, les caractères spéciaux tels que **#**, **@** etc. sont interdits, à l'exception du caractère **_** (underscore). Le tiret **-** est bien sûr interdit puisqu'il correspond aussi à la soustraction.
- Mots réservés : Python se réserve un certain nombre de mots comme **for**, **while**, **class**, **function**.... Ces mots ne peuvent être des noms de variable.

Noms de variables

- Le nom d'une variable est composé de lettre **a** à **z**, **A** à **Z**, et de chiffres **0** à **9**, mais ne doit pas commencer par un chiffre.
- Seules les lettres ordinaires sont autorisées. Les lettres accentuées, les cédilles, les espaces, les caractères spéciaux tels que **#**, **@** etc. sont interdits, à l'exception du caractère **_** (underscore). Le tiret **-** est bien sûr interdit puisqu'il correspond aussi à la soustraction.
- Mots réservés : Python se réserve un certain nombre de mots comme **for**, **while**, **class**, **function**.... Ces mots ne peuvent être des noms de variable.
- En Python, les variables sont créées automatiquement à leur première utilisation. Pour créer une variable, il suffit donc de l'utiliser en l'affectant une première fois, c'est à dire d'écrire :

```
nom_variable = valeur_de_la_variable
```

Exemples de déclarations de variables

```
1 a
```

Donne

```
-----  
NameError                                Traceback  
<ipython-input-4-60b725f10c9c> in <module>()  
----> 1 a  
  
NameError: name 'a' is not defined
```

Mais

```
1 a = 5  
2 a
```

donne

```
5
```

Opérateurs

Vocabulaire :

- Un *opérateur* est un symbole utilisé pour effectuer un calcul entre un ou des opérandes.

Opérateurs

Vocabulaire :

- Un *opérateur* est un symbole utilisé pour effectuer un calcul entre un ou des opérandes.
- Une *opérande* est une variable ou un littéral ou bien une expression.

Opérateurs

Vocabulaire :

- Un *opérateur* est un symbole utilisé pour effectuer un calcul entre un ou des opérandes.
- Une *opérande* est une variable ou un littéral ou bien une expression.
- Une *expression* est une suite valide d'opérateurs et d'opérandes. Dans `x+y` : deux opérandes, un opérateur.

Opérateurs

Types d'opérateurs :

- les opérateurs logiques

Les opérateurs peuvent avoir des sens différents en fonction des types d'opérandes sur lesquels ils agissent.

Opérateurs

Types d'opérateurs :

- les opérateurs logiques
- les opérateurs de comparaisons

Les opérateurs peuvent avoir des sens différents en fonction des types d'opérandes sur lesquels ils agissent.

Opérateurs

Types d'opérateurs :

- les opérateurs logiques
- les opérateurs de comparaisons
- les opérateurs sur les séquences

Les opérateurs peuvent avoir des sens différents en fonction des types d'opérandes sur lesquels ils agissent.

Opérateurs

Types d'opérateurs :

- les opérateurs logiques
- les opérateurs de comparaisons
- les opérateurs sur les séquences
- les opérateurs numériques

Les opérateurs peuvent avoir des sens différents en fonction des types d'opérandes sur lesquels ils agissent.

Opérateurs

Types d'opérateurs :

- les opérateurs logiques
- les opérateurs de comparaisons
- les opérateurs sur les séquences
- les opérateurs numériques
- les opérateurs d'affectation

Les opérateurs peuvent avoir des sens différents en fonction des types d'opérandes sur lesquels ils agissent.

Opérateurs logiques

Les expressions avec un opérateur logique sont évaluées à **True** ou **False** :

- **X or Y** : OU logique (binaire), si X évalué à **True**, alors l'expression est **True** et Y n'est pas évalué. Sinon, l'expression est évaluée à la valeur booléenne de Y.

Opérateurs logiques

Les expressions avec un opérateur logique sont évaluées à **True** ou **False** :

- **X or Y** : OU logique (binaire), si X évalué à **True**, alors l'expression est **True** et Y n'est pas évalué. Sinon, l'expression est évaluée à la valeur booléenne de Y.
- **X and Y** : ET logique (binaire), si X est évalué à **False**, alors l'expression est **False** et Y n'est pas évalué. Sinon, l'expression est évaluée à la valeur booléenne de Y.

Opérateurs logiques

Les expressions avec un opérateur logique sont évaluées à **True** ou **False** :

- **X or Y** : OU logique (binaire), si X évalué à **True**, alors l'expression est **True** et Y n'est pas évalué. Sinon, l'expression est évaluée à la valeur booléenne de Y.
- **X and Y** : ET logique (binaire), si X est évalué à **False**, alors l'expression est **False** et Y n'est pas évalué. Sinon, l'expression est évaluée à la valeur booléenne de Y.
- **not X** : évalué à la valeur booléenne opposée de X. Unaire.

Priorités

- `A or B and C` se comprend comme `A or (B and C)`.
Python évalue d'abord le « and » alors qu'il est à droite.

Priorités

- `A or B and C` se comprend comme `A or (B and C)`.
Python évalue d'abord le « and » alors qu'il est à droite.
- `not A and B` se comprend comme `not(A) and B`.
Python évalue d'abord le « not ».

Priorités

- `A or B and C` se comprend comme `A or (B and C)`.
Python évalue d'abord le « `and` » alors qu'il est à droite.
- `not A and B` se comprend comme `not(A) and B`.
Python évalue d'abord le « `not` ».
- On dit que la priorité de `not` est plus grande que celle de `and`, elle-même plus grande que celle de `or`.

Opérateurs de comparaisons

Tout comme les opérateurs logiques, les opérateurs de comparaison renvoient une valeur booléenne **True** ou **False**. Les opérateurs de comparaisons s'appliquent sur tous les types de base.

- `<` strictement inférieur

Opérateurs de comparaisons

Tout comme les opérateurs logiques, les opérateurs de comparaison renvoient une valeur booléenne **True** ou **False**. Les opérateurs de comparaisons s'appliquent sur tous les types de base.

-  strictement inférieur
-  strictement supérieur

Opérateurs de comparaisons

Tout comme les opérateurs logiques, les opérateurs de comparaison renvoient une valeur booléenne **True** ou **False**. Les opérateurs de comparaisons s'appliquent sur tous les types de base.

- `<` strictement inférieur
- `>` strictement supérieur
- `<=` inférieur ou égal

Opérateurs de comparaisons

Tout comme les opérateurs logiques, les opérateurs de comparaison renvoient une valeur booléenne **True** ou **False**. Les opérateurs de comparaisons s'appliquent sur tous les types de base.

- `<` strictement inférieur
- `>` strictement supérieur
- `<=` inférieur ou égal
- `>=` supérieur ou égal

Opérateurs de comparaisons

Tout comme les opérateurs logiques, les opérateurs de comparaison renvoient une valeur booléenne **True** ou **False**. Les opérateurs de comparaisons s'appliquent sur tous les types de base.

- `<` strictement inférieur
- `>` strictement supérieur
- `<=` inférieur ou égal
- `>=` supérieur ou égal
- `==` égal

Opérateurs de comparaisons

Tout comme les opérateurs logiques, les opérateurs de comparaison renvoient une valeur booléenne **True** ou **False**. Les opérateurs de comparaisons s'appliquent sur tous les types de base.

- `<` strictement inférieur
- `>` strictement supérieur
- `<=` inférieur ou égal
- `>=` supérieur ou égal
- `==` égal
- `!=` différent

Opérateurs de comparaisons

Tout comme les opérateurs logiques, les opérateurs de comparaison renvoient une valeur booléenne **True** ou **False**. Les opérateurs de comparaisons s'appliquent sur tous les types de base.

- `<` strictement inférieur
- `>` strictement supérieur
- `<=` inférieur ou égal
- `>=` supérieur ou égal
- `==` égal
- `!=` différent
- `X is Y` : teste si X et Y représentent le même objet.

Opérateurs de comparaisons

Tout comme les opérateurs logiques, les opérateurs de comparaison renvoient une valeur booléenne **True** ou **False**. Les opérateurs de comparaisons s'appliquent sur tous les types de base.

- `<` strictement inférieur
- `>` strictement supérieur
- `<=` inférieur ou égal
- `>=` supérieur ou égal
- `==` égal
- `!=` différent
- `X is Y` : teste si X et Y représentent le même objet.
- `X is not Y` : teste si X et Y ne représentent pas le même objet.

Remarque

Possibilité d'enchaîner les opérateurs comme dans : `X < Y < Z`, dans ce cas, c'est Y qui est pris en compte pour la comparaison avec Z et non pas l'évaluation de `(X < Y)` comme on pourrait s'y attendre dans d'autres langages.

Opérateurs numériques

FIGURE – Opérateurs numériques

symbole	effet	exemple
+	addition	$6+4 == 10$
-	soustraction	$6-4 == 2$
*	multiplication	$6*4 == 24$
/	division	$6/4 == 1.5$
**	élévation à la puissance	$12**2 == 144$
//	division entière	$6//4 == 1$
%	reste de la division entière	$6\%4 == 2$

Comportement de la division différent en Python2.7,

Sémantique des opérateurs

- La *sémantique* (signification) des opérateurs dépend du type des opérandes.

Sémantique des opérateurs

- La *sémantique* (signification) des opérateurs dépend du type des opérandes.
- Exemple avec `+` et `*` :

```
1 print (3+5) # addition d'entier
2 print ([10,30]+[90]) #concaténation de listes
3 print('salut'+ ' les amis') #concaténation de chaînes
4 print (5*3) # produit d'entiers
5 print ([5]*3) #opération externe sur les listes
6 print ([5]+[5]+[5]) #même résultat que ci-dessus
```

Sémantique des opérateurs

- La *sémantique* (signification) des opérateurs dépend du type des opérandes.
- Exemple avec + et * :

```
1 print (3+5) # addition d'entier
2 print ([10,30]+[90]) #concaténation de listes
3 print('salut'+ ' les amis') #concaténation de chaînes
4 print (5*3) # produit d'entiers
5 print ([5]*3) #opération externe sur les listes
6 print ([5]+[5]+[5]) #même résultat que ci-dessus
```

- Donne

```
8
[10, 30, 90]
salut les amis
15
[5, 5, 5]
[5, 5, 5]
```

Opérateurs d'affectation

= opérateur d'affectation :

```
1 x=y=1#affectation parallèle
2 print('x=',x, ' et y=',y)
3 x,y=5,6.8#affectations multiples
4 print('x=',x, ' et y=',y)
```

Donne

```
x= 1    et y= 1
x= 5    et y= 6.8
```

Ordres de priorités

FIGURE – Précédences de la plus faible à la plus forte. Python évalue d'abord « and » puis « or »

Operator	Description
lambda	Lambda expression
if - else	Conditional expression
or	Boolean OR
and	Boolean AND
not x	Boolean NOT
in, not in, is, is not, <, <=, >, >=, <>, !=, ==	Comparisons, including membership tests and identity tests
	Bitwise OR
^	Bitwise XOR
&	Bitwise AND
<<, >>	Shifts
+, -	Addition and subtraction
*, /, //, %	Multiplication, division, remainder [8]
+x, -x, ~x	Positive, negative, bitwise NOT
**	Exponentiation [9]
x[index], x[index:index], x(arguments...), x.attribute	Subscription, slicing, call, attribute reference
(expressions...), [expressions...], {key: value...}, `expressions...`	Binding or tuple display, list display, dictionary display, string conversion

Expressions

- Une *expression* est une suite de caractères définissant une *valeur*. Pour calculer cette valeur, la machine doit *évaluer* l'expression.
- Par exemple, lorsqu'on entre au clavier la chaîne de caractères `4+5` et qu'on appuie sur (SHIFT+) ENTER, Python effectue la somme de quatre et cinq (il *interprète* une chaîne de caractères et retourne un résultat).

État courant

- Une variable est une représentation idéale d'une zone mémoire de l'ordinateur. Il s'agit d'un endroit où l'on peut stocker une valeur, y accéder et changer cette valeur.

État courant

- Une variable est une représentation idéale d'une zone mémoire de l'ordinateur. Il s'agit d'un endroit où l'on peut stocker une valeur, y accéder et changer cette valeur.
- L'ensemble des variables définies à un instant donné de l'exécution d'un programme est appelé un *état courant de l'exécution d'un programme* (état courant) ou encore *environnement*.

État courant

- Une variable est une représentation idéale d'une zone mémoire de l'ordinateur. Il s'agit d'un endroit où l'on peut stocker une valeur, y accéder et changer cette valeur.
- L'ensemble des variables définies à un instant donné de l'exécution d'un programme est appelé un *état courant de l'exécution d'un programme* (état courant) ou encore *environnement*.
- Une expression peut prendre une valeur en fonction de l'état courant et une autre dans un autre environnement.

```
1 x=3 #environnement (x,3)
2 print(x+4)# une 1ere valeur pour x+4
3 x=5 #environnement (x,5)
4 print(x+4)# et une 2nde valeur pour x+4
```

7

9

Instruction

- Ainsi une expression est plus complexe à évaluer qu'une valeur constante : on parle des *valeurs possibles d'une expression*.
- L'état courant est un ensemble de couples (nom, valeur) comme $\{(a, 3); (x, 5)\}$.
- On appelle *instruction* un ordre de modification de l'état courant.
- `a, x = 3, 5` est une instruction (en fait 2 instructions) car elle modifie l'environnement.
- `x+a` est une expression, elle ne modifie pas l'environnement.

Types d'instructions

- On connaît déjà la déclaration et l'affectation.

Types d'instructions

- On connaît déjà la déclaration et l'affectation.
- Les instructions *composées* :

Types d'instructions

- On connaît déjà la déclaration et l'affectation.
- Les instructions *composées* :
 - **la séquence** qui permet d'exécuter deux instructions l'une à la suite de l'autre,

Types d'instructions

- On connaît déjà la déclaration et l'affectation.
- Les instructions *composées* :
 - **la séquence** qui permet d'exécuter deux instructions l'une à la suite de l'autre,
 - **le test** ou instruction conditionnelle qui permet de n'exécuter une instruction que dans certains états,

Types d'instructions

- On connaît déjà la déclaration et l'affectation.
- Les instructions *composées* :
 - **la séquence** qui permet d'exécuter deux instructions l'une à la suite de l'autre,
 - **le test** ou instruction conditionnelle qui permet de n'exécuter une instruction que dans certains états,
 - **la boucle** qui permet d'exécuter plusieurs fois la même instruction dans un programme.

Séquences

- La manière la plus simple d'assembler deux instructions consiste à les placer l'une à la suite de l'autre. L'exécution du programme suit cet ordre.

```
1 a=1
2 a=a+6
3 print(a) # qu'affiche cette expression ?
```

Séquences

- La manière la plus simple d'assembler deux instructions consiste à les placer l'une à la suite de l'autre. L'exécution du programme suit cet ordre.

```
1 a=1
2 a=a+6
3 print(a) # qu'affiche cette expression ?
```

- L'ordre des instructions compte ! Que se passe-t-il si on interverti les instructions précédentes ?

Séquences

- La manière la plus simple d'assembler deux instructions consiste à les placer l'une à la suite de l'autre. L'exécution du programme suit cet ordre.

```
1 a=1
2 a=a+6
3 print(a) # qu'affiche cette expression ?
```

- L'ordre des instructions compte ! Que se passe-t-il si on interverti les instructions précédentes ?
- Sémantique de la séquence** Si $i(e)$ est l'état obtenu en exécutant l'instruction i dans l'état e , on a, après l'exécution des deux instructions i_1 puis i_2 , l'état $i_2(i_1(e)) = i_2 \circ i_1(e)$.
Enchaîner deux instructions en séquence revient donc à composer (au sens des fonctions mathématiques) leurs deux actions sur l'état.

Séquences

- La manière la plus simple d'assembler deux instructions consiste à les placer l'une à la suite de l'autre. L'exécution du programme suit cet ordre.

```
1 a=1
2 a=a+6
3 print(a) # qu'affiche cette expression ?
```

- L'ordre des instructions compte ! Que se passe-t-il si on interverti les instructions précédentes ?
- Sémantique de la séquence** Si $i(e)$ est l'état obtenu en exécutant l'instruction i dans l'état e , on a, après l'exécution des deux instructions i_1 puis i_2 , l'état $i_2(i_1(e)) = i_2 \circ i_1(e)$.
Enchaîner deux instructions en séquence revient donc à composer (au sens des fonctions mathématiques) leurs deux actions sur l'état.
- Il est possible de considérer des séquences de plus de deux instructions : on parle alors de *bloc d'instructions*.

Test simple

Une instruction conditionnelle est une instruction pouvant s'exécuter seulement si une condition est vérifiée par l'état courant.

- Syntaxe :

```
1 if condition:  
2     bloc_d_instructions
```

Test simple

Une instruction conditionnelle est une instruction pouvant s'exécuter seulement si une condition est vérifiée par l'état courant.

- Syntaxe :

```
1 if condition:
2     bloc_d_instructions
```

- Par exemple :

```
1 if x % 2 == 1:
2     x = x + 1
```

Cette instruction ajoute 1 à **x** seulement si **x** est impair.
Si **x** est pair : l'instruction ne modifie pas **x**

Test avec alternative

- On veut ajouter 1 à `x` quand la variable est impaire et la diviser par 2 sinon, pourrait-on écrire ce qui suit ?

```
1 if x % 2 == 1: # si x est impair...
2     x = x + 1 # il est modifié ici...
3 if not(x % 2 == 1): # et comme il est devenu pair...
4     x = x // 2 # il est modifié une seconde fois !
```

Test avec alternative

- On veut ajouter 1 à `x` quand la variable est impaire et la diviser par 2 sinon, pourrait-on écrire ce qui suit ?

```
1 if x % 2 == 1: # si x est impair...
2     x = x + 1 # il est modifié ici...
3 if not(x % 2 == 1): # et comme il est devenu pair...
4     x = x // 2 # il est modifié une seconde fois !
```

- Réponse non : si `x` est impair, on lui ajoute 1 (elle devient paire)
PUIS on le divise par 2.
Ce n'est pas ce qu'on veut.

Test avec alternative

- On veut ajouter 1 à `x` quand la variable est impaire et la diviser par 2 sinon, pourrait-on écrire ce qui suit ?

```
1 if x % 2 == 1: # si x est impair...
2     x = x + 1 # il est modifié ici...
3 if not(x % 2 == 1): # et comme il est devenu pair...
4     x = x // 2 # il est modifié une seconde fois !
```

- Réponse non : si `x` est impair, on lui ajoute 1 (elle devient paire)
PUIS on le divise par 2.
Ce n'est pas ce qu'on veut.
- On a besoin du *Test avec alternative* (**si ... alors... sinon...**) :

```
1 if condition:
2     bloc_d_instruction_si_condition_verifiee
3 else:
4     bloc_d_instruction_si_condition_pas_verifiee
```

Test avec alternative

Réponse pour l'exemple ci-dessus :

```
1 if x % 2 == 1:
2     x = x + 1
3 else:
4     x = x // 2 # x n'est modifiée qu'une fois !
```

Tests imbriqués

- On peut réaliser une instruction conditionnelle dans une instruction conditionnelle : *tests imbriqués*.
- Exécuter trois instructions différentes selon la valeur de $x \% 3$

```
1 if x % 3 == 0:
2     x = x + 1 # cette instruction semble privilégiée
3 else:
4     if x % 3 == 1:
5         x = x - 1 # pkoi indenter cette instruction ?
6     else:
7         x = 2 * x
```

Tests imbriqués

Le mot clé `elif`

Plus élégant :

```
1 if x % 3 == 0:
2     x = x + 1
3 elif x % 3 == 1:
4     x = x - 1
5 else:
6     x = 2 * x #toutes les instructions au mm niveau.
```

Importance de l'indentation

Quelle est la différence entre

```
1 x,y=6,7
2 if x%2 == 1:
3     print('impair')
4 y = -2
5 print(y)
```

et

```
1 x,y=6,7
2 if x%2 == 1:
3     print('impair')
4     y = -2
5 print(y)
```

Exercice

Exercice

Écrire un programme qui détermine si un automobiliste est en excès de vitesse, connaissant sa vitesse et le type de voie sur lequel il circule (donné sous forme d'une chaîne de caractères : « agglomération, route, autoroute »).

Exercice

Exercice

Que fait ce programme ?

```
1 if a>b:
2     if a>c:
3         m = a
4     else:
5         m = c
6 else:
7     if b>c:
8         m = b
9     else:
10        m = c
```

Exercice

Exercice

Le programme suivant permet de déterminer si un individu est en surpoids, par le biais du calcul de son indice de masse corporelle.

```
1 masse = float(input("Quelle est votre masse en kg ?"))
2 taille = float(input("Quelle est votre taille en m ?"))
3 imc = masse / taille**2
4 if imc > 25 :
5     print("Vous etes en surpoids.")
```

Adapter ce programme pour qu'il détermine également si l'individu est en sous-poids, ce qui correspond à un indice de masse corporelle inférieur à 18.

Types de boucles

On distingue deux sortes de boucles :

- Les boucles inconditionnelles : `for` .

A utiliser lorsqu'on connaît à l'avance le nombre d'itérations qu'il faudra effectuer.

Types de boucles

On distingue deux sortes de boucles :

- Les boucles inconditionnelles : `for`.
A utiliser lorsqu'on connaît à l'avance le nombre d'itérations qu'il faudra effectuer.
- Les boucles conditionnelles : `while`.
A utiliser lorsqu'on ne connaît pas à l'avance le nombre d'itérations qu'il faudra effectuer.

Boucle `for`

On connaît le nombre d'itérations à effectuer.

- Syntaxe `for c in range( n):` .

Il y aura `n` itérations à effectuer. Numérotées de `0` à `n-1`.

Boucle `for`

On connaît le nombre d'itérations à effectuer.

- Syntaxe `for c in range( n):` .

Il y aura `n` itérations à effectuer. Numérotées de `0` à `n-1`.

- Calcul de 2^5 :

```
1 p, n = 1, 5
2 # l'instruction suivante parcourt
3 # tous les entiers de 0 à n-1 :
4 for c in range(n):
5     p = 2 * p # noter l'indentation
```

Boucle `for`

On connaît le nombre d'itérations à effectuer.

- Syntaxe `for c in range( n):` .

Il y aura `n` itérations à effectuer. Numérotées de `0` à `n-1`.

- Calcul de 2^5 :

```
1 p, n = 1, 5
2 # l'instruction suivante parcourt
3 # tous les entiers de 0 à n-1 :
4 for c in range(n):
5     p = 2 * p # noter l'indentation
```

- Le compteur de boucle est géré par la boucle `for` . Pas besoin de l'incrémenter.

Exercice

Boucle for

Exercice

Ecrire un programme qui calcule la factorielle d'un entier positif `n` avec une boucle `for`.

Le faire tourner à la main

Exercice

Boucle for

Exercice

Que devrait faire le programme suivant ? Identifier les problèmes d'initialisation. Corriger.

```
1 n = int(input('entrer un nb : ')) #saisir n au clavier
2 for i in range(n):
3     if i % 2 == 0:
4         carre = i ** 2
5     else:
6         carre = 0
7     total = total + carre
8 print total
```

Objets itérables

Parcourir un objet itérable

- Un objet *itérable*, est un objet qu'on peut parcourir élément par élément.

Objets itérables

Parcourir un objet itérable

- Un objet *itérable*, est un objet qu'on peut parcourir élément par élément.
- Exemples : n-uplets, chaînes de caractères, listes.

Objets itérables

Parcourir un objet itérable

- Un objet *itérable*, est un objet qu'on peut parcourir élément par élément.
- Exemples : n-uplets, chaînes de caractères, listes.
- Principe :

```
1 for element in objet_iterable:  
2     traiter(element)
```

Objets itérables

Parcourir un objet itérable

- Un objet *itérable*, est un objet qu'on peut parcourir élément par élément.
- Exemples : n-uplets, chaînes de caractères, listes.
- Principe :

```
1 for element in objet_iterable:  
2     traiter(element)
```

- Pour une chaîne de caractères, l'itération a lieu caractère par caractère.

Objets itérables

Parcourir un objet itérable

- Un objet *itérable*, est un objet qu'on peut parcourir élément par élément.
- Exemples : n-uplets, chaînes de caractères, listes.
- Principe :

```
1 for element in objet_iterable:  
2     traiter(element)
```

- Pour une chaîne de caractères, l'itération a lieu caractère par caractère.
- **Pour information uniquement :**
Il est possible de dériver les classes des types de base pour créer ses propres types itérables.
On peut également fabriquer ses propres types d'objets itérables sans hériter des itérables de base en utilisant le protocole d'itération du langage.

Objets itérables

Exemple de parcours

```
1 print('***** Avec des entiers')
2 for i in [1,2,3]:
3     print(i**2)
4 print('***** Avec une chaîne de caractères')
5 for caractere in 'info':
6     print(caractere*2)
```

```
***** Avec des entiers
1
4
9
***** Avec une chaîne de caractères
ii
nn
ff
oo
```

Objets itérables

Exercice

Calculer la somme des éléments d'une liste de nombres.

Objets itérables

range

- `range(n)` est un itérable mais il a un comportement différent en Python 2.7 et 3.*.

Objets itérables

range

- `range(n)` est un itérable mais il a un comportement différent en Python 2.7 et 3.*.
- Ce qui est commun : `range(2,8)` désigne tous les entiers entre 2 (inclu) et 8 (exclu).

Objets itérables

range

- `range(n)` est un itérable mais il a un comportement différent en Python 2.7 et 3.*.
- Ce qui est commun : `range(2,8)` désigne tous les entiers entre 2 (inclu) et 8 (exclu).
- Python2.7 : `range(3)` est une liste calculée immédiatement au moment de l'appel :

```
1 range(3)
```

```
[0, 1, 2]
```

Objets itérables

range

- `range(n)` est un itérable mais il a un comportement différent en Python 2.7 et 3.*.
- Ce qui est commun : `range(2,8)` désigne tous les entiers entre 2 (inclu) et 8 (exclu).
- Python2.7 : `range(3)` est une liste calculée immédiatement au moment de l'appel :

```
1 range(3)
```

```
[0, 1, 2]
```

- Python3 : le calcul n'est pas immédiat (c'est mieux pour la mémoire). Éléments évalués à la volée.

```
range(3) # non calculé explicitement
```

```
range(0, 3)
```

Boucles conditionnelles

- Syntaxe :

```
1 while condition:  
2     bloc_d_instructions
```

Tant que la condition est vérifiée on effectue l'action.
Il faut faire attention aux boucles infinies !!

Boucles conditionnelles

- Syntaxe :

```
1 while condition:  
2     bloc_d_instructions
```

Tant que la condition est vérifiée on effectue l'action.

Il faut faire attention aux boucles infinies!!

- Calculer 2^n et faire tourner à la main avec `n=5`.

```
1 n=5  
2 p=1  
3 while n>0:  
4     p=2*p  
5     n= n-1# ne pas oublier de décrémenter  
6 print(p)
```

Boucles conditionnelles

- Syntaxe :

```
1 while condition:
2     bloc_d_instructions
```

Tant que la condition est vérifiée on effectue l'action.

Il faut faire attention aux boucles infinies!!

- Calculer 2^n et faire tourner à la main avec `n=5`.

```
1 n=5
2 p=1
3 while n>0:
4     p=2*p
5     n= n-1# ne pas oublier de décrémenter
6 print(p)
```

- En général, on utilise les boucles `while` quand on ne sait pas à l'avance combien de fois on va entrer dans la boucle.

Boucles conditionnelles

- Syntaxe :

```
1 while condition:  
2     bloc_d_instructions
```

Tant que la condition est vérifiée on effectue l'action.

Il faut faire attention aux boucles infinies!!

- Calculer 2^n et faire tourner à la main avec `n=5`.

```
1 n=5  
2 p=1  
3 while n>0:  
4     p=2*p  
5     n= n-1# ne pas oublier de décrémenter  
6 print(p)
```

- En général, on utilise les boucles `while` quand on ne sait pas à l'avance combien de fois on va entrer dans la boucle.

- Tout ce qu'on sait faire avec `for` on peut le faire avec `while` et

Boucles infinies

- Incrémentation

```
1 i = 0
2 while i >= 0:
3     i = i + 1 # on ne sort jamais de cette boucle
```

Boucles infinies

- Incrémentation

```
1 i = 0
2 while i >= 0:
3     i = i + 1 # on ne sort jamais de cette boucle
```

- Le programme suivant demande d'entrer un nombre jusqu'à ce qu'un mot clé soit tapé :

```
1 stop = False
2 while not stop:
3     n = input("entrer un nombre")
4     print(type(n))
5     if n == "stop":
6         stop = True
7     print("vous avez choisi", n)
```

Exercice : Boucles conditionnelles



Exercice : Boucles conditionnelles

Exercice

Modifier le programme précédent pour qu'il demande d'entrer deux entiers n, k et calcule k^n .



Exercice : Boucles conditionnelles

Exercice

Modifier le programme précédent pour qu'il demande d'entrer deux entiers n , k et calcule k^n .

```
1 k = int(input(" entrer un nombre : "))
2 n = int(input(" entrer un exposant : "))
3 res =1
4 while n>0:
5     res=k*res
6     n= n-1# ne pas oublier de décrémenter
7 print(res)
```

les mots clés `def` et `return`

- Pour créer une fonction ou une procédure : utiliser le mot clé `def` .
- Une *procédure* est une fonction sans retour de valeur.
- Fonction avec retour de valeur, utiliser le mot clé `return`
- Traditionnellement, on parle de *fonction* lorsqu'il y a retour de valeur, sinon on parle de *procédure*. Mais on mélange souvent les deux.

Exemple

```
1 def powerp(n,k):  
2     res=1  
3     while n>0:  
4         res=res*k  
5         n-=1 #n=n-1  
6     print(res) #affiche k^n; pas de retour
```

Exemple

```
1 def powerf(n,k):  
2     res=1  
3     while n>0:  
4         res=res*k  
5         n-=1 #n=n-1  
6     return res #retourne k**n
```

Exemple

```
1 def powerpf(n,k):  
2     res=1  
3     while n>0:  
4         res=res*k  
5         n-=1 #n=n-1  
6     print(n*k) #affiche n*k  
7     return res #retourne k**n
```

Vocabulaire

```
1 def somme(a,b):  
2     return a+b  
3 print ("on calcule la somme de 7 et 8")  
4 print(somme(7,8))
```

- Première ligne : *déclaration de la fonction* ou *signature*
- Bloc d'instructions après les deux points : *corps* de la fonction.
- a,b : paramètres *formels*.
- 7,8 : paramètres *effectifs*.
- somme(7,8) est un *appel* de la fonction somme.
- Tout ce qui n'est pas indenté constitue le *programme principal*.

L'objet **None**

- Une fonction qui *semble* ne rien retourner (absence de **return**) retourne néanmoins une valeur particulière : **None**.
- Un exemple :

```
1 >>> def coucou(n):  
2 ...     print('coucou'*n)  
3 ...  
4 >>> coucou(3)  
5 coucoucoucoucoucou  
6 >>> None is coucou(3)  
7 coucoucoucoucoucou  
8 True
```

Fonctions anonymes

- On utilise le mot clé `lambda` pour introduire une fonction *anonyme*.

```
1  >>> def f(x):  
2  ...     return x*2  
3  ...  
4  >>> f(3)  
5  6  
6  >>> g = lambda x: x*2  
7  >>> g(3)  
8  6  
9  >>> (lambda x: x*2)(3)  
10 6
```

Application d'une fonction à une liste

- On utilise le mot clé `map` pour appliquer une fonction à tous les éléments d'une liste.

```
1 def f(x):  
2     return x**2  
3 t=[1,2]  
4 print(f(t))#une erreur TypeError est déclenchée
```

- Avec `map`

```
1 print(map(f,t))# affiche <map object at 0x7f2cac323f2  
2 print(list(map(f,t)))#affiche [1,4]
```

Paramètres optionnels

On peut déclarer des paramètres ayant une valeur par défaut. Si on accepte la valeur par défaut, il n'est pas besoin de citer le paramètre lors de l'appel.

```
1 def f(a,b,c=0, d=1):  
2     return (a/b)**c*d  
3  
4 print(f(4,2))# (4/2)^0=1.  
5 print(f(4,2,3))# (4/2)^3=8.
```

Le premier paramètre ayant une valeur par défaut doit apparaître dans la déclaration après le dernier sans valeur par défaut

Paramètres optionnels

Ordre de passage des paramètres

```
1 def f(a,b,c=0, d=1):  
2     return (a/b)**c*d  
3  
4 print(f(4,2,d=0)) #0.  
5 print(f(4,2,d=2,c=3)) #16.0  
6 print(f(d=2,b=4,a=4,c=0)) #2.0
```

Importer

- Importer une fonction d'un module :

```
1 from math import sqrt
2 sqrt(4)
```

2.0

- Importer tout un module :

```
1 import time
2 time.localtime()
```

```
time.struct_time(tm_year=2016, tm_mon=9, tm_mday=
tm_min=19, tm_sec=55, tm_wday=3, tm_yday=252, tm
```

Préfixe obligatoire.

Importer

- Pas de préfixe mais conflits de noms possibles :

```
1 from fractions import *  
2 Fraction(5,6)+Fraction(1,4)
```

```
Fraction(13, 12)
```

- Préfixe choisi :

```
1 import sympy as s  
2 x = s.var('x')  
3 s.solve(x**2 - 5*x + 6, x)
```

```
[2, 3]
```

Travailler avec l'interpréteur

L'interpréteur est ouvert depuis un terminal dans le répertoire **Dupont**. Dans ce répertoire se trouve un fichier **factorielle.py** contenant une fonction **facto**.

- En Python 2.7 :

```
1 >>> import factorielle as f #une fois et une seule
2 >>> f.facto(5)
3 120
4 >>> reload(f) # Recharger, lorsqu'on a fait des modifs
5 <module 'factorielle' from 'factorielle.pyc'>
```

On effectuera **reload(f)** autant de fois qu'on modifiera **factorielle.py**.

Travailler avec l'interpréteur

L'interpréteur est ouvert depuis un terminal dans le répertoire **Dupont**. Dans ce répertoire se trouve un fichier **factorielle.py** contenant une fonction **facto**.

- En Python 3 :

```
1 >>> import factorielle as f # une seule fois
2 >>> f.facto(5)
3 120
4 >>> import imp # une seule fois
5 >>> imp.reload(f) # autant de fois que de modifs
6 <module 'factorielle' from 'factorielle.py'>
```

Travailler avec un IDLE comme **Spyder**

- Ouvrir IDLE.
- Aller dans **File** → **Open**
- Taper **factorielle**. Une nouvelle fenêtre s'ouvre.
- Aller dans la 2ème fenêtre. Appuyer sur la touche **F5**. Le module est chargé !
- Pour toute modification dans **factorielle**, retaper **F5**.
- En fait, l'interpréteur gère pour nous l'importation et le rechargement.

Données mutables et immuables

- Une donnée *mutable*, est une donnée qu'on peut modifier. Après modification, l'objet conserve la même adresse mémoire (= `id`).
Exemple : quand je vais chez le coiffeur, je change de look mais je conserve mon ADN.

Données mutables et immuables

- Une donnée *mutable*, est une donnée qu'on peut modifier. Après modification, l'objet conserve la même adresse mémoire (= `id`).
Exemple : quand je vais chez le coiffeur, je change de look mais je conserve mon ADN.
- A l'inverse, une donnée *immutable* ne peut pas être modifiée. On peut avoir l'impression que c'est possible, mais après la modification, on ne retrouve pas la donnée elle même, seulement une copie pas tout à fait conforme de l'original.

Données mutables et immuables

- Une donnée *mutable*, est une donnée qu'on peut modifier. Après modification, l'objet conserve la même adresse mémoire (= `id`).
Exemple : quand je vais chez le coiffeur, je change de look mais je conserve mon ADN.
- A l'inverse, une donnée *immutable* ne peut pas être modifiée. On peut avoir l'impression que c'est possible, mais après la modification, on ne retrouve pas la donnée elle même, seulement une copie pas tout à fait conforme de l'original.
- Liste de types immuables : `int float decimal complex bool string tuple range fr`

Données mutables et immuables

- Une donnée *mutable*, est une donnée qu'on peut modifier. Après modification, l'objet conserve la même adresse mémoire (= `id`).
Exemple : quand je vais chez le coiffeur, je change de look mais je conserve mon ADN.
- A l'inverse, une donnée *immutable* ne peut pas être modifiée. On peut avoir l'impression que c'est possible, mais après la modification, on ne retrouve pas la donnée elle même, seulement une copie pas tout à fait conforme de l'original.
- Liste de types immuables : `int float decimal complex bool string tuple range`
- Liste de types mutables : `list dict set bytearray user-defined classes (unless`

La méthode spéciale `__iadd__`

Pour connaître la liste des attributs et méthode associés à un objet `o`, entrer `dir(o)`. Une longue liste est alors affichée.

```
1 x = 3
2 print(type(x))
3 print(dir(x))
```

```
<class 'int'>
['__abs__', '__add__', '__and__', '__bool__',
 '__ceil__',.... PAS LA PLACE DE TOUS LES METTRE]
```

Pour vérifier que les entiers sont immutables, on cherche s'ils possèdent la méthode `__iadd__` (« je m'ajoute à moi-même »)

```
1 print('__iadd__' in dir(x)) #False : immutable
```

```
False
```

La méthode spéciale `__iadd__`

Pour connaître la liste des attributs et méthode associés à un objet `o`, entrer `dir(o)`. Une longue liste est alors affichée.

```
1 x = [5,6] #une liste
2 print(type(x))
3 print(dir(x))
```

```
<class 'list'>
['__add__', '__class__', '__contains__', '__delattr__',
 '__delitem__', '__dir__', '__doc__', '__eq__', ...
PAS LA PLACE DE TOUS LES METTRE]
```

Pour vérifier que les listes sont mutables, on cherche si elles possèdent la méthode `__iadd__` (« je m'ajoute à moi-même »)

```
1 print('__iadd__' in dir(x)) # True : les listes sont mutables
```

```
True
```

Données mutables

Exemples avec Python

- Données persistantes : Si je mange trop de chocolat, c'est mon clone qui grossit.
-

Données mutables

Exemples avec Python

- Données persistantes : Si je mange trop de chocolat, c'est mon clone qui grossit.

En Python, les (entiers, tuples, flottants...) sont des implémentations de données persistantes et sont dites *immuables*.

```
1 a = 5
2 print(id(a))# le numéro de a
3 a += 7
4 print(id(a))# a a changé
```

```
140060647799968
```

```
140060647800192
```



Données mutables

Exemples avec Python

- Données persistantes : Si je mange trop de chocolat, c'est mon clone qui grossit.
- Données non persistantes : Si je mange trop de chocolat, c'est moi qui grossi pas mon clone.

Données mutables

Exemples avec Python

- Données persistantes : Si je mange trop de chocolat, c'est mon clone qui grossit.
- Données non persistantes : Si je mange trop de chocolat, c'est moi qui grossi pas mon clone. Les listes et dictionnaires sont des implémentations de données persistantes et sont dites *mutables*.

```
1 t = [5]
2 print(id(t)) # le numéro de t
3 t += [7]
4 print(id(t)) # t a grossi
```

```
140059818168584
140059818168584
```

Un peu plus sur les listes

```
1 t = [50,30,40,10]
2 # 1er, second, dernier, avant dernier elt
3 print(t[0], t[1], t[-1], t[-2])
4 # tous les éléments du 1er (inclus) au dernier (exclu)
5 print(t[1:-1])
6 M = [[1,2,3],[4,5,6]]
7 print(M[1][2])
8 print(M[1])
```

```
50 30 10 40
[30, 40]
6
[4, 5, 6]
```

Un peu plus sur les listes

```
1 t = [50,30,40,10]
2 print(t)
3 print(id(t))
4 t.append(0) #t grossit; les listes sont mutables
5 print(t)
6 print(id(t))
7 t+=[1] #même effet que append
8 print(t)
9 print(id(t))
```

```
[50, 30, 40, 10]
140709375714696
[50, 30, 40, 10, 0]
140709375714696
[50, 30, 40, 10, 0, 1]
140709375714696
```

Un peu plus sur les listes

```
1 t = [50,30,40,10]
2 print(t)
3 print(id(t))
4 t+=[1] # toujours le même t
5 print(t)
6 print(id(t))
7 t = t + [1] # un autre t
8 print(t)
9 print(id(t))
```

```
[50, 30, 40, 10]
140599714627272
[50, 30, 40, 10, 1]
140599714627272
[50, 30, 40, 10, 1, 1]
140599486834632
```

Tuples vs listes

```
1 t=(1,2,3)
2 print(t[2])#affiche 3
3 t[1] = 5#soulève une exception car t n'est pas mutable
```

Mais

```
1 t=(1,2,3)
2 print(t)
3 print(id(t))
4 t+=(5,6)#une copie de t grossit
5 print(t)
6 print(id(t))#t n'est pas mutable
```

```
(1, 2, 3)
140709435130432
(1, 2, 3, 5, 6)
140709375758848
```

Un peu plus sur `range`

```
1 print(range(5,8))
2 print(list(range(5,8)))
3 print(list(range(5,10,3)))
4 print(list(range(10,7,-1)))
```

```
range(5, 8)
[5, 6, 7]
[5, 8]
[10, 9, 8]
```

Listes par compréhension

```
1 #carrés des entiers entre 1 et 4
2 print ([x**2 for x in range(1,5)])
3 #carrés des entiers impairs entre 1 et 4
4 print ([x**2 for x in range(1,5) if x%2==1])
5 # une matrice de VanderMonde
6 print([[j ** i for j in range(5)] for i in range(4)])
```

```
[1, 4, 9, 16]
```

```
[1, 9]
```

```
[[1, 1, 1, 1, 1], [0, 1, 2, 3, 4],
```

```
[0, 1, 4, 9, 16], [0, 1, 8, 27, 64]]
```

Fonctions et objets immutables

```
1 n=1#variable globale
2 def f(n):
3     n += 1#c'est une copie de n qui est modifiée
4     print(n)
5 f(n)
6 print(n)#n a conservé sa valeur car immutable
```

2

1

Fonctions et objets mutables

```
1 n=[1] #variable globale
2 def f(n):
3     n += [1]
4     print(n)
5 f(n)
6 print(n) #n a été modifié car mutable
```

```
[1, 1]
```

```
[1, 1]
```

Variables globales et locales

```
1 n=1#variable globale
2 def f(a,b):
3     #a, b paramètres formels
4     c=a+b#c est locale
5     c=c+n#n, globale, existe encore dans f
6 f(0,0)
7 print(c)# c n'existe pas hors de f
```

```
-----
NameError                                Traceback
<ipython-input-22-9b22a643799b> in <module>()
      6     print(c)
      7 f(0,0)
----> 8 print(c)# c n'existe pas hors de f

NameError: name 'c' is not defined
```

Variables déclarées globales

```
1 c=1
2 def f(a):
3     global c#c est déclarée globale
4     c=a#c est modifiée
5 f(0)
6 print(c)
```

0

Variables globales et locales

- Dans le corps d'une fonction, si une variable `y` apparaît à droite d'un opérateur d'affectation (comme dans `x=y+1`) sans être déclarée avant dans le corps de la fonction, cette variable est globale.

Variables globales et locales

- Dans le corps d'une fonction, si une variable `y` apparaît à droite d'un opérateur d'affectation (comme dans `x=y+1`) sans être déclarée avant dans le corps de la fonction, cette variable est globale.
- Si une variable `x` immutable apparaît à gauche d'un opérateur d'affectation (comme dans `x=y+1`), cette variable est locale. Les modifications de `x` dans le corps de la fonction n'affecteront pas la valeur d'une éventuelle variable `x` dans le programme principal.

Variables globales et locales

- Dans le corps d'une fonction, si une variable `y` apparaît à droite d'un opérateur d'affectation (comme dans `x=y+1`) sans être déclarée avant dans le corps de la fonction, cette variable est globale.
- Si une variable `x` immutable apparaît à gauche d'un opérateur d'affectation (comme dans `x=y+1`), cette variable est locale. Les modifications de `x` dans le corps de la fonction n'affecteront pas la valeur d'une éventuelle variable `x` dans le programme principal.
- En revanche, pour une variable mutable `t` , l'écriture `t+=q` dans le corps de la fonction change `t` même au niveau du programme principal.

Variables globales et locales

- Dans le corps d'une fonction, si une variable `y` apparaît à droite d'un opérateur d'affectation (comme dans `x=y+1`) sans être déclarée avant dans le corps de la fonction, cette variable est globale.
- Si une variable `x` immutable apparaît à gauche d'un opérateur d'affectation (comme dans `x=y+1`), cette variable est locale. Les modifications de `x` dans le corps de la fonction n'affecteront pas la valeur d'une éventuelle variable `x` dans le programme principal.
- En revanche, pour une variable mutable `t`, l'écriture `t+=q` dans le corps de la fonction change `t` même au niveau du programme principal.
- Par contre, pour une variable mutable `t`, l'écriture `t=t+q` dans le corps de la fonction ne change pas `t` au niveau du programme principal.

Affectation et instruction

- Dans le programme officiel « On peut signaler par exemple que le fait que l'affectation soit une instruction est un choix des concepteurs du langage Python et en expliquer les conséquences. »

Affectation et instruction

- Dans le programme officiel « On peut signaler par exemple que le fait que l'affectation soit une instruction est un choix des concepteurs du langage Python et en expliquer les conséquences. »
- En Python, l'affectation avec `=` est une *instruction* (statement), pas une expression. Autrement dit, elle ne produit pas de valeur exploitable là où on attend une expression.

Conséquence 1 — Pas d'affectation dans une expression

- En Python, = est une **instruction**, pas une expression.
- Donc impossible dans une condition, une expression ou une lambda.

```
if (x = f()):           # SyntaxError
    ...

y = (x = 1)            # SyntaxError

lambda: x = 1          # SyntaxError
```

Conséquence 2 — Affectations chaînées

- L'affectation chaînée est possible, mais reste une **instruction**.

```
x = y = 0          # OK : x et y réfèrent le même objet
z = (x = y = 0)    # SyntaxError : ce n'est pas une expression
```

Conséquence 3 — Affectations augmentées

- `x += 1` etc. sont aussi des **instructions**, pas des expressions.

```
x = 1
x += 1                # OK

y = (x += 1)          # SyntaxError : pas utilisable
```

Conséquence 4 — Besoin ponctuel : l'opérateur `:=`

- Depuis Python 3.8, `:=` (walrus) permet une **affectation en expression**, sans changer la nature de `=`.

```
# Lecture jusqu'à ligne vide
while (line := f.readline()):
    ...

# Affectation + test pattern
if (m := pattern.search(s)):
    print(m.group(0))
```