Introduction à la Programmation des Algorithmes

3. Python

Semaine 1 : Notions de base

Automne 2025

Dr Julia Buwaya

UNIVERSITÉ
DE GENÈVE

3. Python

Semaine 1 : Notions de base

1 Hello World

Nous avons vu dans 2. Introduction que le langage Python est **interprété**, c'est à dire que le fichier source sera lu par un **interpréteur Python** qui est un exécutable.

Si le fichier `hello-world.py` contient le code source suivant :

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

exécuter dans un terminal

```
python3 hello-world.py
```

affiche

```
Hello World!
```

Si le fichier `hello-world.py` contient le code source suivant :

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

exécuter dans un terminal

```
python hello-world.py
```

affiche

```
Hello World!
```

L'interpréteur Python détecte seul automatiquement s'il y a un problème et indique une erreur.

```
1 prin("Hello World!")
```

```
python test.py
  File "test.py", line 1
    prin("Hello World!")
    ^^^^
NameError: name 'prin' is not defined. Did you mean: 'print'?
```

Il est possible de rajouter des commentaires qui seront ignorés lors de l'exécution du programme. On indique qu'une partie de ligne est un commentaire en la préfixant avec un `#`.

```
1 print(3)
2
3 # je viens d'afficher 3 et je vais afficher 42!
4
5 print(42)
```

2 Types

Les types de base sont :

- `int` représente un entier,
- `float` représente un réel,
- `str` représente une Sting (chaîne de caractères),
- `bool` représente une valeur booléenne.

Python possède de manière native des types complexes pour gérer des listes ou des collections. Nous y reviendrons ultérieurement.

Python possède un type supplémentaire `NoneType` qui ne peut prendre que la valeur `None` et permet par exemple de représenter une grandeur qui n'est pas encore disponible.

Python possède donc de manière native des types dont la manipulation demandent des opérations sophistiquées en mémoire (allocation, copies) et cache cette complexité au programmeur.

Il permet donc en particulier de manipuler les Stings (chaînes de caractères) de façon infiniment plus simple qu'en C.

Python possède donc de manière native des types dont la manipulation demandent des opérations sophistiquées en mémoire (allocation, copies) et cache cette complexité au programmeur.

Il permet donc en particulier de manipuler les Stings (chaînes de caractères) de façon infiniment plus simple qu'en C.

 Cette complexité sous-jacente fait que Python est un langage très lent. En pratique les opérations coûteuses sont faites à l'aide de librairies écrites dans un langage de bas niveau comme le C.

3 Strings

Les **Strings (chaînes de caractères)** sont des suites de caractères qui peuvent être manipulées littéralement, c'est à dire que l'interpréteur/le compilateur ne donne pas de sens particulier à ce qu'elles contiennent.

Une chaîne de caractères est délimitée dans le programme par le caractère " par ex. `"ceci est une chaîne de caractères"`

On peut mettre certains caractères spéciaux en les préfaçant avec \, par ex. `"cette chaîne revient\nà la ligne"`, `"et celle ci contient un \""`

Les Strings en Python sont encodées en Unicode dont le standard actuel (08.2021) inclut 140'000 caractères, dont toutes les lettres accentuées, les alphabets chinois, cyrillique, etc. ainsi que les émojis.

Une chaîne littérale en Python peut être délimitée avec " ou '.

```
In [1]: a='Une chaîne normale, avec un accent'

In [2]: b='Слава Україні!'

In [3]: c='🤗🍾'

In [4]: print(a,b,c)
Une chaîne normale, avec un accent Слава Україні! 🤗🍾

In [5]: d=a+' / '+b+' / '+c

In [6]: print(d)
Une chaîne normale, avec un accent / Слава Україні! / 🤗🍾

In [ ]:
```

Il est possible de noter une chaîne de caractères littérale sur plusieurs lignes en la délimitant avec """.

```
a = """ceci
est une chaînes sur
plusieurs lignes!!!
"""
```

Il est possible de noter une chaîne de caractères littérale sur plusieurs lignes en la délimitant avec """.

```
a = """ceci
est une chaînes sur
plusieurs lignes!!!
"""
```

Et la fonction `len` retourne la longueur d'une chaîne

```
>>> len("123456")
6
```

Python possède des opérateurs de concaténation de chaînes de caractères `+` et `+=`.

```
1  a = "le chat"
2  b = "chasse"
3  c = "la souris"
4
5  s = a + " " + b
6  print(s)
7
8  s += " " + c
9  print(s)
```

affiche

```
le chat chasse
le chat chasse la souris
```

De plus l'opérateur [] permet d'extraire un caractère (en réalité une sous-chaîne de longueur 1) d'une chaîne de caractères

```
1 s = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
2 print(s[0], s[1], s[2], s[24], s[25])
```

affiche

```
A B C X Y
```

Cet opérateur permet également d'extraire une sous-chaîne à l'aide du symbole : pour spécifier les indexes du premier caractères à prendre et l'indexe qui suit le dernier caractère à inclure.

Si le premier est absent, la valeur par défaut est 0.

Si le second est absent, la valeur par défaut est la longueur de la chaîne complète.

Ces indexes peuvent être négatifs, auquel cas la longueur de la chaîne est automatiquement ajoutée, ce qui permet d'indexer en partant de la fin.

```
1  s = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
2  t = s[3:5]
3  u = s[:5]
4  r = s[16:]
5  v = s[16:-2]
6  print(t, u, r, v)
```

affiche

```
DE ABCDE QRSTUVWXYZ QRSTUVWX
```

4 print() et input()

Nous utiliserons dans les exemples la fonction `print` qui affiche simplement ses arguments dans l'ordre, suivis d'un retour à la ligne.

Nous verrons plus tard comment faire des formatages plus sophistiqués.

L'existence des chaînes de caractères gérées automatiquement permet en particulier de faire facilement des saisis.

La fonction `input` retourne une chaîne de caractères entrée interactivement par l'utilisateur.

```
1 nom = input("Entrez votre nom: ")
2 print("Bonjour chère/cher", nom)
```

affiche (en vert ce que tape l'utilisateur)

```
Entrez votre nom: Francois
Bonjour chère/cher Francois
```

5 Variables

Une **variable** est créée lors de la première **affectation**. De plus son **type** peut changer à la suite d'une autre affectation.

La fonction `type` retourne le type d'une expression.

```
1  a = 3
2  print(a, type(a))
3  a = "toto"
4  print(a, type(a))
```

affiche

```
3 <class 'int'>
toto <class 'str'>
```

Il est possible de faire plusieurs affectations d'un coup en séparant les variables et leurs valeurs par des virgules :

```
1  a, b = 3, 4
2  print(a, b)
```

affiche

```
3 4
```

 Le symbole `'='` a un sens très différent de celui qu'il a en mathématique. Il ne définit pas une propriété qui est et sera toujours vraie. Il indique simplement de changer quelque chose en mémoire : "copie la valeur spécifiée à droite dans la variable spécifiée à gauche."

Un **identifiant** est un label composé de chiffres, lettres minuscules, lettres majuscules et du "underscore", c'est à dire le caractère `'_'`. Un identifiant ne peut pas commencer par un chiffre.

Exemples :

- `n`
- `x0`
- `nombre_de_voitures`
- `compteurDeTours`
- `_tmp`

Les **mots-clés** sont réservés pour le langage lui-même et ne peuvent pas être utilisés comme identifiants, par exemple

```
str
int
float
print
if
else
for
```

Nous en verrons plusieurs dans ce cours.

Les **constantes** sont des valeurs numériques qui apparaissent littéralement dans le programme.

Il y a de nombreux points de syntaxe que nous n'allons pas détailler et nous nous limiterons aux entiers, par ex. `34`, ou `87934`, et aux valeurs à virgules, possiblement en notation scientifique, par ex. `3.1415926`, `-45.0`, ou `7.8e9`.

Les **constantes** sont des valeurs numériques qui apparaissent littéralement dans le programme.

Il y a de nombreux points de syntaxe que nous n'allons pas détailler et nous nous limiterons aux entiers, par ex. `34`, ou `87934`, et aux valeurs à virgules, possiblement en notation scientifique, par ex. `3.1415926`, `-45.0`, ou `7.8e9`.

L'interpréteur/le compilateur considère qu'une constante numérique sans `.` est de type entier, et qu'une avec est de type à virgule.

 Une constante entière qui commence par `0` est interprétée comme étant notée en base 8.

6 Conversion de type

Python fait des conversions de types numériques implicitement pour garder la précision autant que possible :

```
1  a = 3
2  b = 4.2
3  print(a, type(a))
4  print(b, type(b))
5  print(a + b, type(a + b))
```

affiche

..

```
3 <class 'int'>
4.2 <class 'float'>
7.2 <class 'float'>
```

Et Python permet des conversions explicites avec des fonctions associées aux différents types.

```
1  x = 3
2  print(x, type(x))
3
4  x = float(x)
5  print(x, type(x))
6
7  x = str(x)
8  print(x, type(x))
9
10 x = float(x)
11 print(x, type(x))
```

affiche

```
3 <class 'int'>
3.0 <class 'float'>
3.0 <class 'str'>
3.0 <class 'float'>
```

La manière la plus simple et standard de formater des valeurs à afficher consiste à utiliser une chaîne de caractères formatée, aussi appelées *f-string*.

Il suffit de la préfacer par `f` et de spécifier les expressions à substituer entre `{}` dans le corps de la chaîne.

```
1  a = 3
2  b = 4.5
3  c = "Bob"
4
5  print(f"Nous avons {a} et {a + b} et aussi {c}")
```

affiche

```
Nous avons 3 et 7.5 et aussi Bob
```

La manière la plus simple et standard de formater des valeurs à afficher consiste à utiliser une chaîne de caractères formatée, aussi appelées *f-string*.

Il suffit de la préfacer par `f` et de spécifier les expressions à substituer entre `{}` dans le corps de la chaîne.

```
1  a = 3
2  b = 4.5
3  c = "Bob"
4
5  print(f"Nous avons {a} et {a + b} et aussi {c}")
```

affiche

```
Nous avons 3 et 7.5 et aussi Bob
```

Il est possible de définir plus précisément le format (notation scientifique, nombre de chiffre après la virgule, etc.)

7 Opérateurs et expressions

Une expression est une séquence d'opérateurs et d'opérandes qui définit un calcul à effectuer. Une opérande est un des arguments d'un opérateur.

Par exemple

```
3 + 2 * x
```

est une expression valide en Python qui combine les opérandes `3`, `2`, et `x`, avec les opérateurs `+` et `*`.

Une expression est une séquence d'opérateurs et d'opérandes qui définit un calcul à effectuer. Une opérande est un des arguments d'un opérateur.

Par exemple

```
3 + 2 * x
```

est une expression valide en Python qui combine les opérandes `3`, `2`, et `x`, avec les opérateurs `+` et `*`.

Étant données les règles standard de priorités, nous comprenons cette expression comme

```
3 + ( 2 * x )
```

Contrairement aux expressions dans un langage naturel, celles dans un langage de programmation ont un sens (une “sémantique”) exacte, qui correspond à la suite d’instructions de langage machine que l’ordinateur exécutera.

Les **opérateurs arithmétiques**

`+`	addition
`-`	soustraction
`*`	multiplication
`/`	division (en virgule flottante)
`//`	division euclidienne
`%`	modulo (reste de la division)

L'opérateur `/` fait toujours le calcul en virgule flottante, et Python a `//` pour la division euclidienne.

```
1 a = 14
2 b = 5
3 print(a / b, a // b)
```

affiche

```
2.8 2
```

Les opérateurs `+ -` ont une priorité inférieure à `* / // %`. Dans le cas d'opérateurs de même priorité, le calcul est effectué de gauche à droite.

Les opérateurs d'**affectation composée**, sont des opérateurs binaires qui modifient leur opérande de gauche.

	Opération	Valeur
`n += k`	`n = n + k`	`n` après le changement
`n -= k`	`n = n - k`	`n` après le changement
`n *= k`	`n = n * k`	`n` après le changement
`n /= k`	`n = n / k`	`n` après le changement
`n %= k`	`n = n % k`	`n` après le changement

8 Limitations et dangers des types numériques

 Comme nous avons vu, les types numériques souffrent de limitations de représentation. Par exemple

```
1  print(150 * 1.12)
```

affiche

```
168.00000000000003
```

au lieu de 168

Fin

julia.buwaya@unige.ch