Outils de simulation

HAP603P, Faculté des Sciences de Montpellier, 2022
Felix Brümmer (felix.bruemmer@umontpellier.fr)

Introduction

Dans ce chapitre

Généralités

- Programmation scientifique et physique numérique
- Aperçu du langage Python

Python

- Instructions

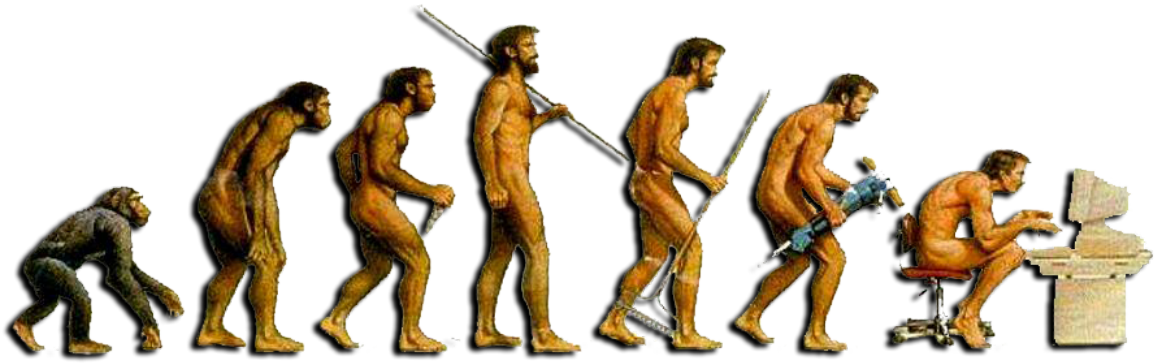
Pourquoi physique numérique ?

L'ordinateur est un outil indispensable en physique :

- Solution numérique des équations qui décrivent les systèmes physiques
- Simulation des systèmes complexes
- Assistance aux calculs analytiques
- Analyse et traitement des données
- Visualisation des résultats, rédaction des publications scientifiques
- ...

Pourquoi apprendre à programmer ?

- Développer des programmes.
- Modifier et adapter à ses besoins des programmes et bibliothèques existants.
- Connaitre le mode de fonctionnement et les limites des logiciels que l'on utilise.



Qu'est-ce qu'un programme ?

Un programme est un ensemble de commandes qui amènent l'ordinateur à changer l'état de sa mémoire interne et/ou de ses périphériques.

Typiquement nous commanderons l'ordinateur d'effectuer certaines opérations et d'afficher ou d'enregistrer les résultats.

Un langage de programmation est un ensemble de règles syntactiques que les commandes doivent suivre afin qu'elles puissent être traduites en instructions au système d'exploitation (ou directement au matériel informatique).

Cette traduction aura lieu soit au fur et à mesure lors de l'exécution du programme (interprétation) soit une seule fois avant l'exécution (compilation).

Langages de programmation

```python
# Afficher "Hello World!" avec Python

print("Hello World!")
```

```cpp
// Afficher "Hello World!" avec C++

#include<iostream>
using namespace std;

int main() {
    cout << "Hello World!\n";
    return 0;
}
```

```fortran
! Afficher "Hello World!" avec Fortran 90

PROGRAM HelloWorld
   WRITE (*,*) 'Hello World!'
END PROGRAM HelloWorld
```

Python

Pour ce cours on va se servir du langage de programmation `Python`.

Pourqoui Python ?

- tres répandu
  (devenant le standard universel en programmation scientifique, sauf pour le calcul à haute performance)
- facile à apprendre, syntaxe simple et intuitive
  $\Rightarrow$ permet de se concentrer sur la programmation sans être encombré par les pièges du langage
- polyvalent, de multiples domaines d'application
- beaucoup des bibliothèques incluses, vaste fonctionnalité
- haut niveau d'abstraction (pas de manipulation directe du matériel informatique)
- moderne : inclut des concepts comme la programmation orientée objet, la programmation fonctionnelle, le typage dynamique, la gestion automatique de mémoire. . .

Quelques domaines de la programmation scientifique

Calcul symbolique / calcul formel

- Manipulation automatisée des objets mathématiques au niveau symbolique, c.à.d. sans forcement une application numérique : Algèbre, analyse, arithmétique...
- Les systèmes de calcul formel incluent typiquement des langages de programmation complets pour gérer le flot d'exécution. Pas seulement des "grandes calculatrices" !
- Interfaces typiquement interactives de type notebook (calepin / cahier de travail).
- Exemples : `Mathematica`, `Maple`, `MATLAB`, `SageMath/SymPy`

* propriétaire * basé sur `Python`

Quelques domaines de la programmation scientifique

Calcul numérique à haute performance

- Evaluation numérique des fonctions / des intégrales à haute précision, solution numérique de systèmes d'équations algébriques ou différentielles, optimisation, simulation de systèmes complexes avec un grand nombre de dégrés de liberté. . .
- Domaine classique de l'analyse numérique.
- Les problèmes peuvent être très demandeur côté puissance de calcul : besoin d'optimisation de code (automatisée ou de la part du programmeur) pour exploiter au mieux le matériel informatique.
- On utilise les langages de programmation compilés pour optimisation, souvent de bas niveau d'abstraction pour minimiser l'overhead.
- Interface : soit éditeur + console avec ses outils (compilateur, lieur, débogueur), soit environnement de développement intégré
- Exemples : `FORTRAN`, `C`, `C++`

Quelques domaines de la programmation scientifique

Gestion, traitement, analyse et visualisation des données

- Traitement automatisé des données expérimentales, observationnelles, numériques...
- Stockage, tri, analyse statistique, ajustement, représentation graphique (création de figures ou des animations)...
- Mons besoin de puissance de calcul, plus besoin de fonctionnalités diverses (polyvalence)
- Domaine des langages de programmation interprétés.
- Interface :
  - soit fichiers de script (crée par éditeur ou environnement de développement intégré)
  - soit interface interactif (notebook ou ligne de commande)
- Exemples de langages de script : script shell de Unix, `Perl`, `R`, `Julia`, `Python`
- Exemples de suites logicielles interactives : `MATLAB`, `Octave`, `IPython/Jupyter`

* propriétaire * basé sur `Python`

Quelques domaines de la programmation scientifique

Développement des logiciels et outils auxiliaires

- Programmation des interfaces, de l'environnement graphique, du web, des bases de données, du système d'exploitation...
- Interférences avec d'autres secteurs du développement des logiciels
- Domaine des langages universels, interprétés ou compilés selon l'objectif
- Interface : typiquement environnement de développement intégré
- Exemples : `C++`, `Java`, `Python`

Deux façons d'exécuter son programme : Compilation et interprétation

1. Programmes compilés :

Le code source est entièrement traduit en forme exécutable par un logiciel auxiliaire, le compilateur. Un autre logiciel auxiliaire, l'éditeur de liens, peut assister afin d'intégrer les composantes du programme et les bibliothèques externes dans le fichier exécutable.

Forces :

- Le compilateur peut automatiquement optimiser le code lors de la traduction.
- Pour cette raison les programmes compilés sont souvent plus rapides et efficaces.
- Manque de transparence : les producteurs de logiciels commerciaux peuvent cacher les détails de fonctionnement des programmes compilés.

Faiblesses :

- Après toute modification du code source il faut recompiler.
- Manque de portabilité : chaque système d'exploitation a son propre compilateur et sa propre version du code compilé.
- Pas de possibilité d'exécuter seulement une partie d'un programme
- Manque de transparence : sans le code source, l'utilisateur ne sait pas que fait le programme en détail

Deux façons d'exécuter son programme : Compilation et interprétation

2. Programmes interprétés :

Le code source est traduit ligne par ligne lors de l'exécution par un logiciel auxiliaire, l'interpréteur.

Forces :

- Après une modification du code source on peut immédiatement réexécuter le nouveau programme.
- Les erreurs de programmation sont souvent plus faciles à détecter (débogage)
- Portabilité : les seuls fichiers de programme sont celles du code source, qui sont les mêmes sur tout système

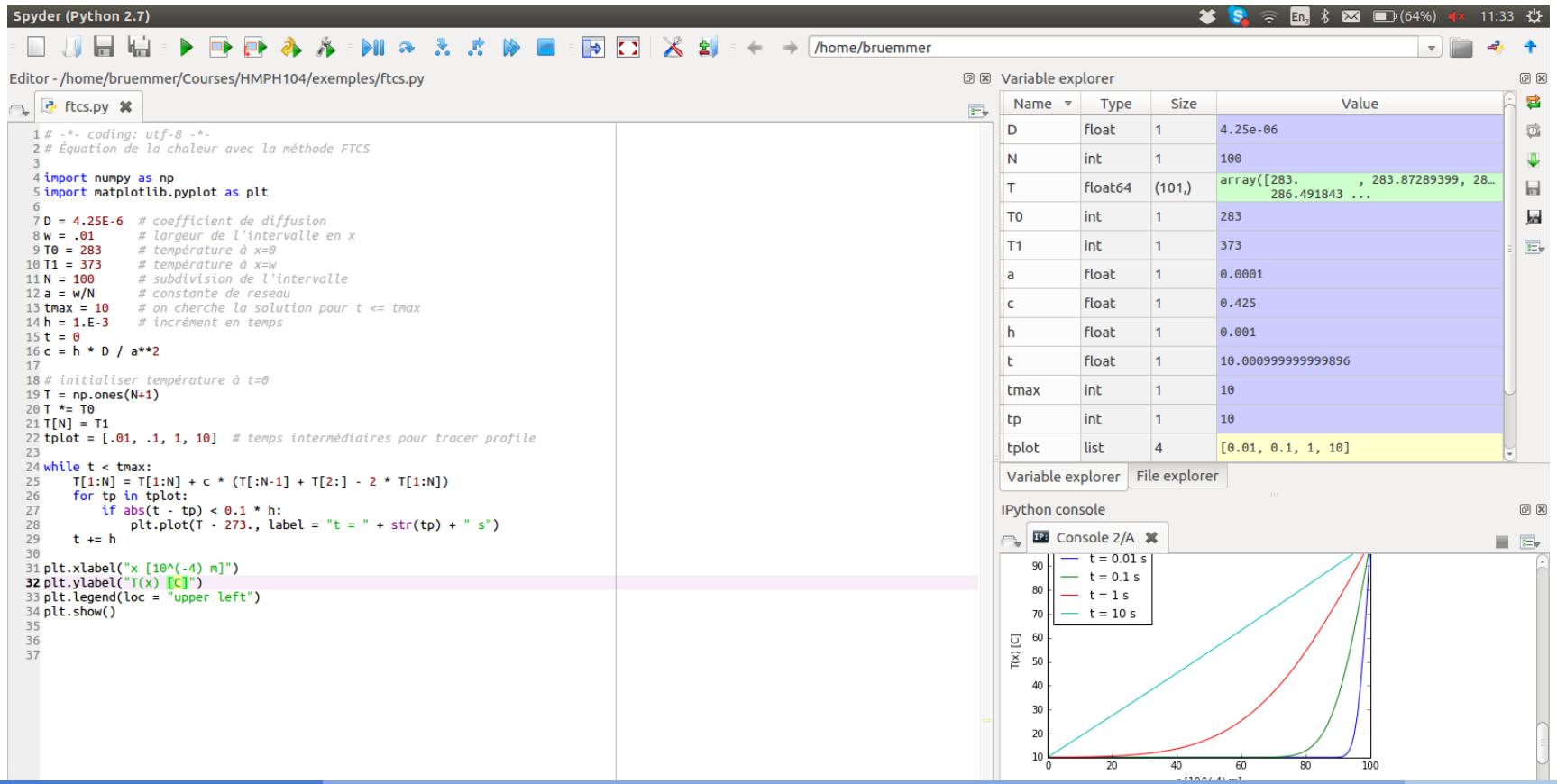
Faiblesses :

- Efficacité généralement réduite par rapport à un programme compilé et optimisé.

Deux façons de rédiger son code : Fichier de code source et notebook

1. Fichier de code classique / script :

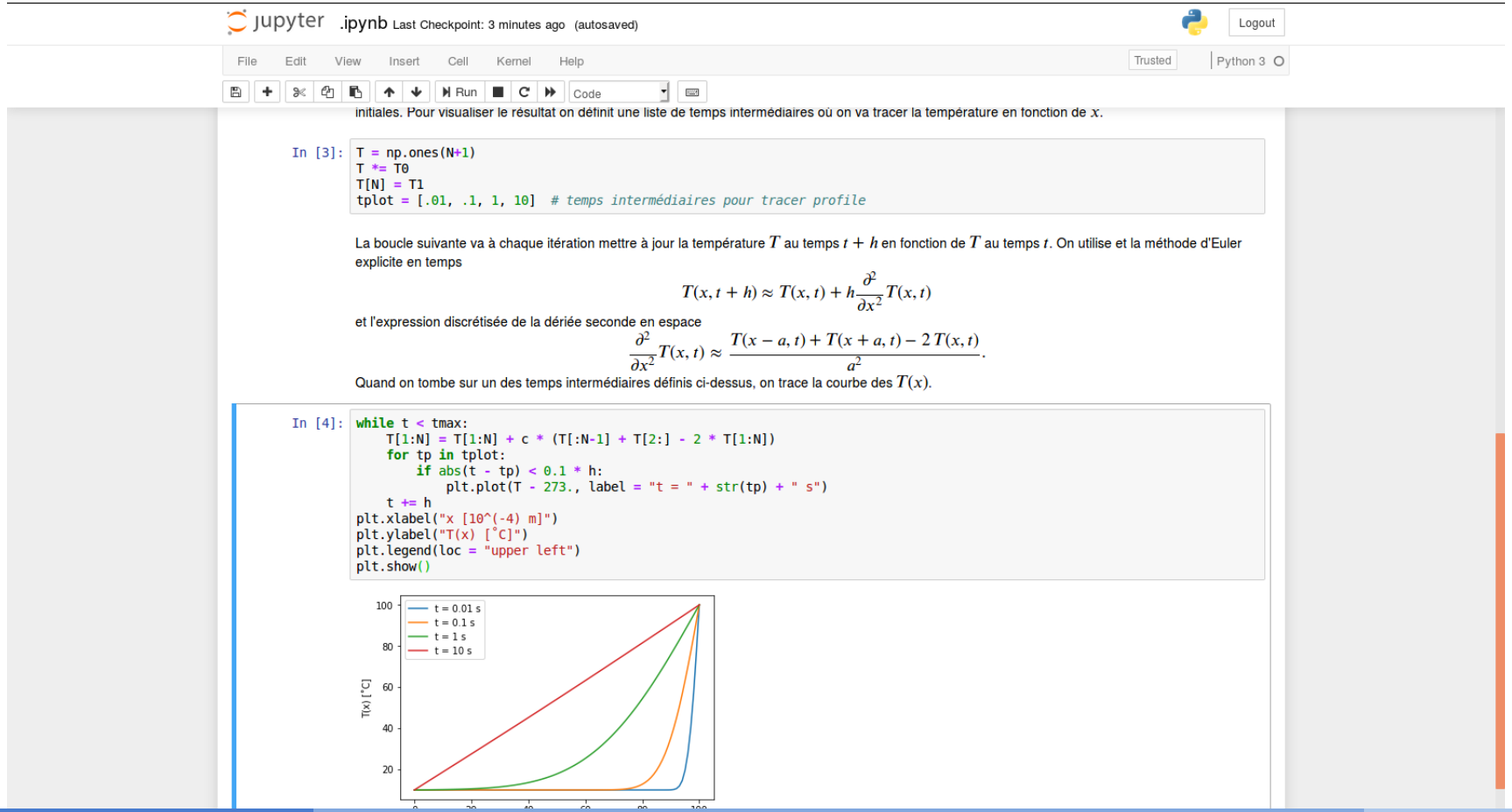
- Créé avec un éditeur de texte (qui peut faire partie d'un environnement de développement intégré).
- Format : fichier de texte, ne contient que le code de source et des commentaires
- Le code est enregistré dans un ou plusieurs fichiers que l'on passe ensuite au compilateur ou à l'interpréteur.



Deux façons de rédiger son code : Fichier de code source et notebook

2. Notebook interactif :

- Créé avec un éditeur dédié.
- Format spécial, peut contenir du texte formaté, des équations, des résultats intermédiaires, des éléments graphiques. . . ainsi que le code de source
- Interpréteur toujours intégré, on peut exécuter le notebook entier ou juste une partie



Deux paradigmes de programmation : Procédurale et orientée objet

1. Programmation procédurale :

- Date des années '50
- Séparation entre les données et les procédures qui les gèrent
- Structure des programmes plus linéaire
- Mieux adaptée aux petits projets car moins d'overhead.
- Exemples de langages bien adaptés à la programmation procédurale : `FORTRAN`, `C`, `Python`

Deux paradigmes de programmation : Procédurale et orientée objet

2. Programmation orientée objet :

- Date des années '80
- Notion centrale : L'objet qui réunit les données et les méthodes, représentant une entité abstraite définie par son état et ses capacités
- Tout objet est instance d'une classe. Il y a une hiérarchie des classes avec des propriétés héritables.
- Structures plus abstraites.
- Programmes moins linéaires, favorisant la modularité
- Mieux adaptée aux grands projets de plusieurs contributaires.
- Exemples de langages bien adaptés à la programmation orientée objet : `C++`, `Java`, `Python`

Python

Le langage Python

- est un langage interprété

  Plus précisement, son implémentation standard CPython traduira le code source en "bytecode" qui est ensuite interprété.

- s'utilise soit en mode de script soit en mode interactif
- permet tant la programmation procédurale que la programmation orientée objet.

Aperçu du cours

Ce cours porte sur des sujets en programmation scientifique et en analyse numérique :

- Révision de la programmation procédurale avec Python
- Méthodes de recherche de zéros
- Calcul matriciel avec NumPy, graphisme
- Résolution numérique d'équations différentielles ordinaires
- Méthodes de l'algèbre linéaire numérique
- Applications : Optimisation ; ajustement ; valeurs propres pour problèmes aux limites
- Si le temps le permet : Introduction au calcul symbolique

Prérequis pour le suivre avec profit :

- Connaissances en programmation avec Python ("Physique sur ordinateur" en L2)...
- ...en physique (mécanique, électrodynamique, mécanique quantique)...
- ...et en mathématiques (nombres complexes, analyse réelle, algèbre linéaire).

Matière à revoir indépendamment si nécessaire !

Bibliographie

Python :

- B. Cordeau et L. Pointal, « Une introduction à Python 3 », `http://perso.limsi.fr/pointal/python:courspython3`
- G. Swinnen, « Apprendre à programmer avec Python 3 », `http://www.inforef.be/swi/python.htm`
- D. Cassagne, « Introduction à Python pour la programmation scientifique », `http://wwww.courspython.com`
- « Python Tutorial », `https://docs.python.org/fr/3/tutorial/`
- beaucoup d'autres sources à trouver en ligne et hors ligne

Algorithmes pour la physique numérique :

- M. Newman, « Computational Physics », 2012 (en anglais)
- W. H. Press, S. Teukolsky, W. Vetterling et B. Flannery, « Numerical Recipes », 3e édition 2007, Cambridge University Press (en anglais et C++)

Matériel du cours sur l'espace pédagogique

Vous trouverez sur Moodle :

- Ces notes de cours
- Tous les exemples de code apparaissant ci-dedans : répertoire `Exemples/`
- Toutes les fiches d'exercices

Exécuter son code avec Python

- Rédiger votre code de source avec l'aide de votre éditeur de texte préféré
- L'enrégistrer dans un fichier, par exemple `exemple.py`
- Exécuter le script avec l'interpréteur Python :
  Entrer `python3 exemple.py` par la console (dans le dossier où se trouve le fichier)

Raccourci : Des environnements de développement intégrés (comme `spyder`) permettent de rédiger le code et de l'exécuter directement par un clic dans l'interface graphique.

Il faudra pourtant que vos programmes soient autonomes (ne dépendent pas des fonctionnalités de `spyder` pour générer ses résultats) !

- Plus tard on va travailler aussi avec l'interface interactif `Jupyter`.

Un premier programme en Python

```
#!/usr/bin/python3
# Ecrit la phrase "Hello World!" sur l'écran

print("Hello World!")
```

Dans cet exemple figurent

- des commentaires : précédés par un croisillon #. Tout ce qu'y fait suite dans la même ligne du fichier est ignoré par l'interpréteur.
  On utilise des commentaires surtout pour rendre son code mieux lisible par les programmeurs (soi-même inclus). Il ne faut pas en économiser !
- une ligne blanche, ignorée par l'interpréteur
- une instruction : dans ce cas, un appel à la fonction `print( )` qui écrit à l'écran la chaîne de caractères dans les parenthèses

Exercice

Exécuter ce script (`HelloWorld.py`)

Format et interprétation du code source

- Chaque ligne du script (à part les lignes blanches et les lignes qui ne contiennent que des commentaires) correspond à une instruction.
- L'interpréteur exécutera toutes les instructions, une par une.
- Si l'interpréteur tombe sur une instruction fautive, le programme s'arrête avec un message d'erreur. Les informations là-dedans peuvent être très utiles pour identifier et réparer le problème.

```
pint("Oups!") # erreur: la fonction s'appelle print()
              #
              # Le programme va s'arrêter avec le message
              #  "NameError: name 'pint' is not defined"
              # qui indique que 'pint' n'est pas défini
```

- Sinon, le programme termine dès qu'il n'y a plus d'instructions à exécuter.

Exceptions de la règle d'une instruction par ligne

- Une instruction avec des parenthèses (ou crochets [ou accolades { pas fermés se poursuit sur les lignes suivantes jusqu'à la clôture.

```
print("Malheureusement ce texte est trop long pour une "
"seule ligne de code source, mais on veut cependant "
"l'afficher dans une seule ligne sur l'écran.")
```

- On peut aussi terminer une ligne avec un anti-slash \ pour indiquer qu'une instruction se poursuit sur la ligne suivante.
- On peut grouper plusiers instructions dans une seule ligne si on les sépare avec un point-virgule ;

```
print("Flying"); print("Circus")
```

Pour améliorer la lisibilité du code il est fortement conseillé de mettre une instruction par ligne et une ligne par instruction si possible.

Les instructions : Erreurs fréquentes

- À la différence de Python 2, `print()` est une fonction en Python 3 — il est impératif d'écrire les parenthèses `()`

```
print("Ca marche")          # ça marche
print "Ca ne marche pas"    # ça ne marche pas
```

- Attention à l'orthographe. En particulier, Python est sensible à la casse (distingue entre les minuscules et les majuscules).
- Si un programme ne fonctionne pas : Examiner le message d'erreur. Il contient des informations utiles sur l'endroit où le programme s'est interrompu (la ligne du code source) et sur le genre d'erreur qui s'est produit.
- Contrairement à beaucoup d'autres langages de programmation, des espaces au début d'une ligne (indentation) ont une signification syntactique et sont à éviter (sauf si l'objectif est de créer un bloc ; voir le chapitre "Structures de contrôle").

Les types de données

Dans ce chapitre

Python

- Les variables et les affectations
- Les types de données numériques
- Les opérations arithmétiques
- Les types de données séquentiels
- Les chaînes de caractères
- La saisie du clavier

Les variables, les types et les affectations

Voici quelques exemples d'affectations qui attribuent des valeurs aux variables :

```
phrase = "Mais non!"
nombre = 25
somme = nombre + 5    # la valeur de 'somme' devient 30
nombre3 = 50.0
liste = ["lundi", "mardi", "mercredi"]
```

- Ici `phrase`, `nombre`, `somme`, `nombre3` et `liste` désignent des variables.
- Toute variable est d'un type qui résulte du format utilisé dans l'affectation. Ici '`phrase`' est du type str (chaîne des caractéres), '`nombre`' et '`somme`' sont du type int (nombres entiers), '`nombre3`' est du type float (nombre flottant) et '`liste`' est du type list (liste d'objets).
- Après initialisation la variable peut être utilisée, cf. l'usage de '`nombre`' dans la troisième ligne ci-dessus. En revanche, une commande comme

```
a = b
```

produit une erreur si la variable `b` n'a pas été donnée une valeur avant.

Les variables

Chaque variable est caractérisée par

- son nom (identifiant)
- son type
- sa valeur

Exemple :

```
ma_variable = 25
```

- Ici l'identifiant est `ma_variable`.
  Un identifiant se compose des lettres `A-Z` et `a - z`, du tiret bas `_` et des chiffres `0 - 9` (sauf pour le premier caractère).
- Le type est déterminé automatiquement à l'initialisation. Ici le type est `int` (nombre entier). Si l'initialisation était `ma_variable = 25.0` le type serait `float` (nombre flottant). Si c'était `ma_variable = "vingt-cinq"` le type serait `str` (chaîne des caractères).
- La valeur est 25, bien sûr.

Les identifiants

- Presque toute combinaison de lettres minuscules et majuscules, chiffres (sauf pour le premier caractère) et tirets bas _ est valable comme identifiant.
- Exception : les mots clé du langage Python qui ont une signification syntactique spéciale, soient `and`, `as`, `assert`, `break`, `class`, `continue`, `def`, `del`, `elif`, `else`, `except`, `False`, `finally`, `for`, `from`, `global`, `if`, `import`, `in`, `is`, `lambda`, `None`, `nonlocal`, `not`, `or`, `pass`, `raise`, `return`, `True`, `try`, `while`, `with`, `yield`.
- Deuxième exception : Il y a certaines fonctions natives dont les noms ne doivent pas être utilisés comme identifiants, par exemple `abs`, `complex`, `float`, `input`, `int`, `list`, `max`, `min`, `print`, `str` (même si c'est techniquement possible, ça va entraîner des conséquences imprévisibles).
- Troisième exception : l'identifiant '`self`' ainsi que tout identifiant qui commence avec un ou deux tirets bas (par exemple, '`__init__`') ont une signification spéciale en programmation orientée objet. Il ne faut pas les utiliser hors leur propre contexte.
- Conseil : Utiliser les identifiants parlants pour les variables importantes pour améliorer la lisibilité du code.

```
x = "2 place Victor Hugo, 75000 Paris, France"   # pas idéal
adresse = "34 rue Jean Moulin, 30000 Nîmes, France"  # mieux
```

Les types de données numériques

type	description
`int`	nombre entier sans limite de taille théorique
`float`	nombre flottant, précision 64 bit soit ≈ 15 décimales
`complex`	nombre flottant complexe correspondant à deux `float`

Les types de données numériques : `int`

- Une variable du type `int` (nombre entier) se crée par une affectation sans point décimal comme

```
x = 25
y = -100
```

- Elle est également le résultat d'un appel à la fonction de conversion `int()` :

```
x = 3.8      # x est du type float avec valeur 3.8
y = int(x)   # y est du type int avec valeur 3

z = int("42") # z est du type int avec valeur 42 et a été
              #  construit partant de la chaîne de
              #  caractères "42"
```

Les types de données numériques : `float`

- Une variable du type `float` (qui représente un nombre à virgule flottante avec valeur absolue entre environ 10^{-300} et 10^{300} ou 0, précision numérique ≈ 15 décimales) se crée par une affectation soit avec point décimal soit en notation scientifique :

```
gamma = -5.77
c = 3E8
hbar = 1.05E-34
```

- Ici la notation '3E8' signifie 3×10^8 et la notation '1.05E-34' signifie 1.05×10^{-34}.
- Le résultat d'un appel à la fonction de conversion `float()` est également un nombre flottant :

```
x = 3           # x est entier avec valeur 3
y = float(x)    # y est un float avec valeur 3.0
```

Les types de données numériques : complex

- Une variable du type `complex` représente un nombre flottant complexe, c.à.d. un `float` pour la partie réelle et un pour la partie imaginaire. Notation en Python avec 'J' pour l'unité imaginaire $i = \sqrt{-1}$:

```
c = 3 + 4J              # le nombre complexe 3 + 4 i
d = -2.5 + 3.E-1J       # le nombre complexe -2.5 + 0.3 i
i = 1J                  # le nombre complexe i
```

- La conversion se fait avec la fonction de conversion `complex()` :

```
s = "3 + 2J"      # une chaîne de caractères
z = complex(s)    # un nombre complexe

re_z = 1.7        # un float
im_z = -2.8       # un deuxième float
z = complex(re_z, im_z)  # le nombre complexe 1.7 - 2.8 i
```

Les opérations arithmétiques

Python connaît les opérations arithmétiques suivants :

`+`	addition	
`-`	soustraction	
`*`	multiplication	
`/`	division réelle	résultat est toujours `float` ou `complex`
`//`	division entière	résultat est `int` si les deux arguments sont `int`, `float` ou `complex` sinon
`%`	reste de la division entière	résultat est `int` si les deux arguments sont `int`, `float` ou `complex` sinon
`**`	puissance	

Exemples :

```
x = 5 + 3      # x est du type int avec valeur 8
y = x / 5      # y est du type float la valeur 1.6
z = x // 5     # z est du type int avec valeur 1
yy = 8 % 5.0     # yy = 3.0 est du type float
xx = 2 ** 3      # xx = 8
zz = 16 * 4 - 2 * (10 + 1)    # zz = 42
```

Les affectations

Une affectation procède en deux pas :

- l'expression à droite du signe = est évaluée
  (calculée en fonction de l'état de la mémoire à cet instant)
- le résultat est affecté à la variable dont le nom figure à gauche du signe =

Cela permet des instructions comme

```
x = 0          # x est du type int, sa valeur est 0
x = x + 1      # augmenter x par 1
```

(la 2nde ligne ne doit pas être confondue avec une équation algébrique !)
ou même

```
x = y = 7      # x et y sont du type int avec valeur 7
```

Les affectations

Combinaison des opérations arithmétiques avec des affectations, pour changer la valeur d'une variable :

```
x = 2    # x est initialement du type int avec valeur 2

x += 1   # ajouter 1 à x: x devient 3
         #  (équivalent: x = x + 1)
x *= 3   # multiplier x par 3: x devient 9
         #  (équivalent: x = x * 3)
x -= 1   # soustraire 1 de x: x devient 8
         #  (équivalent: x = x - 1)
x /= 4   # diviser x par 4: x devient 2.0
         #  (équivalent: x = x / 4)
```

Les types séquentiels ("sequence types")

type	description
`str`	chaîne de caractères
`list`	liste muable d'objets
`tuple`	collection immuable d'objets

Les chaînes de caractères

Le type de données `str` (“string” en anglais) représente des chaînes de caractères.

- Toute partie du code source entre apostrophes ' ou guillemets " est interprétée comme un `str`.

```
phrase = "Mon tailleur est riche."
titre_du_cours = 'Outils de Simulation (HAP603P)'
```

- On peut convertir toute variable numérique en `str` avec la fonction `str()`. On peut convertir un `str` en `int` seulement s'il se compose des chiffres. Similairement, on peut convertir un `str` en `float` seulement si les caractères représentent un nombre flottant en notation Python.

```
nombre = "23"
print(nombre + nombre)    # affiche "2323"
print(int(nombre) + int(nombre)) # affiche "46"
```

Les opérations sur des chaînes de caractères

Avec l'opérateur + on peut composer les chaînes de caractères :

```
age = 12
phrase = "J'ai " + str(age) + " ans."
print(phrase)  # affiche "J'ai 12 ans."
```

Avec [ ] on les indice. Attention : l'indice du premier caractère est toujours 0 !

```
print(phrase[0])      # affiche "J"
print(phrase[3])      # affiche "i"
```

L'opérateur [a:b] retourne la sous-chaîne de caractères à partir de l'indice a (inclu) jusqu'à l'indice b (exclu). Si on ne spécifie pas a et/ou b, on obtient la sous-chaîne à partir du début et/ou jusqu'au bout.

```
print(phrase[1:3])      # affiche "'a"
print(phrase[5:])       # affiche "12 ans"
print(phrase[:6])       # affiche "J'ai 1"
```

Le caractère d'échappement dans une chaîne de caractères

Dans une chaîne des caractères, le anti-slash \ a une rôle spéciale : c'est le caractère d'échappement.

- Dans un string littéral entouré des apostrophes ', les caractères " sont traités comme des caractères réguliers et vice-versa. Si on veut inclure le caractère ' (ou ") dans un string littéral qui commence et finit avec ' (ou "), il doit être précédé par un \ :

```
print("J'ai 12 ans")
print('J\'ai 12 ans')  # même résultat
print("\"Infâme!\" s'exclama Bastien.")
```

- La séquence \n dans une chaîne des caractères force la fin de la ligne.
- Pour explicitement écrire un anti-slash il faut en inclure deux :

```
print("Voici un anti-slash: \\")
```

(On rappelle que, hors d'une chaîne des caractères, le anti-slash indique par contre la continuation d'une instruction sur la ligne suivante)

Faire entrer une chaîne de caractères par le clavier

Pour récuperer des données du clavier on se sert de la fonction `input()` :

```
s = input("Entrez quelque chose:")
print("Vous avez entré \"" + s + "\"")
```

`input()` retourne toujours un str. Si on veut lire des données numériques du clavier, il faut les convertir :

```
s = input("Entrez un nombre entier:")
i = int(s)
print(s + " fois " + s + " font " + str(i**2))
```

Les types séquentiels : `list`

Le type `list` représente une liste d'objets. Toute partie du code source entre des crochets `[ ]` est interprétée comme liste. Les éléments sont séparés par des virgules `,`.

```
nombres_premiers = [2, 3, 5, 7, 11]
mois_hiver = ["décembre", "janvier", "février"]
```

- Les éléments d'une liste ne sont pas forcement du même type. Il peut y avoir des doublons.

```
liste_bizarre = [2, 2.0, 2 + 0J, "deux"]
liste_nulle = [0, 0, 0]
```

- On peut même construire des listes dont les éléments sont des listes :

```
matrice3x2 = [[1, 2, 3], [6, 5, 4]]
```

Opérations sur les listes

Comme pour les `str` :
Avec l'opérateur + on peut joindre une liste à une autre :

```
mois = ["Janvier", "Février", "Mars"]
mois2 = ["Avril", "Mai"]
print(mois + mois2)
# affiche "['Janvier', 'Février', 'Mars', 'Avril', 'Mai']"
```

Avec [ ] on les indice. Attention : l'indice du premier élément est toujours 0 !

```
print(mois[1])     # affiche "Février"
print(mois[0][0]) # première lettre du premier élément = "J"
```

L'opérateur `[a:b]` retourne la sous-liste entre l'indice `a` (inclu) et l'indice `b` (exclu). Si on ne spécifie pas a (ou b), on obtient la sous-liste à partir du début (ou jusqu'au bout).

```
print(mois[:1])   # affiche "['Janvier']"
print(mois2[:])   # affiche une copie de toute la liste mois2
```

Les types séquentiels : `tuple`

Le type `tuple` représente une collection d'objets similaire à une liste, mais avec une différence importante : Les `tuple` sont immuables, ils ne peuvent pas être modifiés après initialisation. En pratique ils sont moins utilisés que les listes.

Un `tuple` est crée par des parenthèses `( )` avec les éléments séparés par des virgules `,`. On peut même supprimer les parenthèses en absence d'ambiguïté. Exemples d'utilisation :

```
t = (1, 2, 3)   # crée un tuple
u = 1, 2, 3     # le même tuple
print(u[2])     # affiche "3"

a, b = 1, 2     # affecte les valeurs 1 à a et 2 à b

jour = 21
print("Nous sommes le", jour, "janvier")
# Cette commande affiche le tuple se composant du str
#  "Nous sommes le", de l'int 21 et du str "janvier"
```

Les variables et les types de données : Erreurs fréquentes

- Orthographe : `ma_var`, `Ma_var` et `mavar` sont trois identifiants différents.
- Effectuer une opération, puis ne rien faire avec le résultat ne sert à rien :

```
nombre = 5
nombre * 3  # calcule 5 * 3 et oublie le résultat
```

- Ne pas oublier de convertir ses variables au bon type :

```
age = input("Votre age: ") # input() retourne un str
nais = 2022 - age # erreur: faut convertir age en int
```

- L'indice du premier objet dans une séquence est 0. Si la séquence contient n objets, l'indice du dernier est alors $n - 1$.

```
ma_liste = ["un", "deux", "trois"]
print(ma_liste[1])  # affiche "deux"
print(ma_liste[3])  # erreur: pas de 4-ème élément
```

Les structures de contrôle

Dans ce chapitre

Python

- Les blocs d'instructions
- La structure conditionnelle
- Les expressions logiques
- La priorité des opérateurs
- La boucle `while`
- La boucle `for`

Les blocs d'instructions

Un bloc est une séquence de lignes d'instructions distinguées par leur indentation (décalage par rapport aux lignes qui les entourent). Une ou plusieurs lignes consécutives décalées au même niveau constituent un bloc. Un bloc est toujours précédé par une ligne d'en-tête qui se termine avec un deux-points :

```
...
LIGNE EN-TETE:                  # introduit un bloc
    INSTRUCTION 1
    INSTRUCTION 2
    ...
    DERNIERE INSTRUCTION        # ici le bloc se termine
INSTRUCTION SUIVANTE            # <- ne fait plus partie du bloc
...
```

Les structures de controle permettent d'exécuter toutes les instructions d'un bloc plusieurs fois, ou de les exécuter seulement en fonction d'une condition.

Les blocs d'instructions

Un bloc peut contenir d'autres :

```
...
LIGNE EN-TETE:      # ici commence un bloc
    INSTRUCTION
    INSTRUCTION
    ...
    LIGNE EN-TETE: # ici commence un sous-bloc
        INSTRUCTION
        ...
        DERNIERE_INSTRUCTION  # fin du sous-bloc
    INSTRUCTION     # le 1er bloc se poursuit
    ...
    DERNIERE INSTRUCTION  # fin du 1er bloc
...
```

De même pour les sous-sous-blocs etc.

La structure conditionnelle

La structure conditionnelle (ou structure `if`) prend la forme suivante :

```
if CONDITION:
    INSTRUCTION 1    # bloc à exécuter si CONDITION vérifiée,
    INSTRUCTION 2    #  à sauter sinon
    ...
    DERNIERE INSTRUCTION  # ici le bloc se termine
CONTINUER_ICI    # en tout cas le programme reprend ici
...
```

- Ici `CONDITION` est une expression logique de valeur `True` (vrai) ou `False` (faux).
- Le bloc d'instructions suivant est exécuté seulement si `CONDITION` est `True`.
- Sinon le programme saute le bloc et continue après à `CONTINUER_ICI`.
- Les deux-points : après `CONDITION` font partie de la structure et ne doivent pas être omis

La structure conditionnelle

Exemples :

```
i = int(input("Entrez un nombre entier: "))
if i < 0:
    i = -i  # un bloc qui ne contient qu'une seule ligne
print("La valeur absolue de ce nombre est", i)
```

```
print("Tu veux savoir un sécret?")
reponse1 = input("Entre 'o' si oui: ")
if reponse1 == "o":
    reponse2 = input("T'es sur? Entre 'o' si oui:")
    if reponse2 == "o":
        print("Le voici:\nLa cuillère n'existe pas.")
```

Parenthèse : Les expressions logiques

Le type de données `bool`

Ce type de données représente la valeur booléenne d'une expression logique. Les variables du type `bool` ne peuvent prendre que deux valeurs différentes : `True` (vrai) ou `False` (faux).

On peut définir des variables booléennes de la même manière que des variables numériques, par exemple

```
flag = True
...
if flag:
    FAIRE_QUELQUE_CHOSE
    ...
```

Par la fonction `bool()` on peut convertir un `str` en `bool` (s'il s'agît de la chaîne de caractères `"True"` ou `"False"`). De même pour une variable numérique (dans ce cas le résultat est `False` si le nombre est 0 et `True` sinon). (On peut même directement utiliser la valeur numérique correspondante dans une structure conditionnelle au lieu de la condition – normalement déconseillé car peu lisible.)

Les expressions logiques

Les opérateurs de comparaison :

expression	`True` si ...
`x == y`	`x` est égal à `y`
`x != y`	`x` est différent de `y`
`x > y`	`x` est strictement supérieur à `y`
`x < y`	`x` est strictement inférieur à `y`
`x >= y`	`x` est supérieur ou égal à `y`
`x <= y`	`x` est inférieur ou égal à `y`

Les opérateurs logiques : soient `a` et `b` du type bool (`True` ou `False`)

`not a`	`True` si `a` est `False` et vice-versa
`a and b`	`True` si `a` est `True` et `b` est `True`, `False` autrement
`a or b`	`True` si au moins un de `a` ou `b` est `True`, `False` autrement

Les opérateurs `in` et `is`

L'opérateur `in`

teste si un objet est contenu dans une séquence :

```
animaux = ["giraffe", "gazelle", "guépard"]
"giraffe" in animaux   # True
"gorille" in animaux   # False
"elle" in "gazelle"    # True
```

L'opérateur `is`

teste si deux identifiants désignent le même objet (il ne teste pas l'égalité des valeurs) :

```
x = [1, 2]      # une liste avec deux entrées
y = [1, 2]      # une autre liste avec les mêmes entrées
z = x     # z est un autre nom pour x
x is y    # False
x is z    # True
```

Cet opérateur peut parfois donner des résultats inattendus sur des variables immuables (types numériques, `str`...). On comprendra plus tard pourquoi.

Fin de parenthèse : La priorité des opérateurs

En ordre ascendant :

- `or`
- `and`
- `not`
- comparaisons : `==`, `!=`, `>`, `<`, `>=`, `<=`, `in`, `is`
- addition et soustraction : `+`, `-`
- multiplication et division : `*`, `/`, `//`, `%`
- signe : `+x`, `-x`
- exponentiation : `**`

Ainsi l'expression "`not x > y or - x ** y + 2 * y == 0`" est interprétée

$$(\neg(x > y)) \vee (((-(x^y)) + (2 \times y)) = 0)$$

On peut toujours insérer des parenthèses pour changer les priorités :
"`(-x) ** (y + 2) * y == 0`" devient

$$((-x)^{y+2} \times y) = 0$$

La structure conditionnelle augmentée

Ajouter un bloc `else` (“sinon”), à exécuter si la condition `CONDITION` est `False` :

```
if CONDITION:
    INSTRUCTION # si CONDITION est True
    ...
else:
    AUTRE_INSTRUCTION  # si CONDITION est False
    ...
CONTINUER_ICI          # en tout cas on reprend ici
```

Exemple :

```
i = int(input("Entrez un nombre entier:"))
if i % 2 == 0:
    print(i, "est pair")
else:
    print(i, "est impair")
```

La structure conditionnelle augmentée

Ajouter des blocs `elif` (“sinon, si”) :

```
if CONDITION1:
    INSTRUCTION # si CONDITION1 est True
    ...
elif CONDITION2:
    AUTRE_INSTRUCTION  # si CONDITION1 est False
    ...                #    mais CONDITION2 est True
elif CONDITION3:       # etc.
    ENCORE_AUTRE_INSTRUCTION
    ...
else:
    DERNIERE_CHANCE    # si toutes CONDITIONs sont False
    ...
...
```

La boucle `while`

La boucle `while` (“tant que”) sert à répéter les instructions d'un bloc en fonction d'une condition :

```
while CONDITION:
    FAIRE_QUELQUE_CHOSE # ce bloc est répété tant que
    ...                 #   CONDITION est True
CONTINUER_ICI           # Après on arrive ici
...
```

Exemple :

```
i = 1
while i % 2 != 0:    # condition remplie si i est impair
    i = int(input("Entrez un nombre pair:"))
print("La moitié de", i, "est", i // 2)
```

La boucle `while`

Deuxième exemple : Conjecture de Collatz.

On définit la suite (n_k) par un $n_0 \in \mathbb{N}$ et la règle de récurrence

$$n_{k+1} = \begin{cases} \frac{n_k}{2}\,, & n_k \text{ pair} \\ 3n_k + 1\,, & n_k \text{ impair} \end{cases}$$

Conjecture : Pour toute valeur de départ n_0 on va ultérieurement tomber sur $n_i = 1$ (et puis $n_{i+1} = 4$, $n_{i+2} = 2$, $n_{i+3} = 1$ etc.)

En supposant que la conjecture soit vraie (sinon : boucle infinie, le programme ne terminera jamais !), on calcule le nombre minimal d'itérations i pour tomber sur $n_i = 1$, avec n_0 fourni par l'utilisateur :

```
n = int(input("Entrez n0: "))
i = 0    # compteur d'itérations
while n != 1: # pourvu que la conjecture soit vraie!
    if n % 2 == 0:   # n pair:
        n /= 2       # remplacer n <- n/2
    else:            # n impair:
        n *= 3       # remplacer n <- 3n + 1
        n += 1
    i += 1
print("i =", i)
```

La boucle `for`

La boucle `for` sert à répéter les instructions d'un bloc une fois pour chaque élément d'une séquence :

```
for VAR in SEQUENCE:
    FAIRE_QUELQUE_CHOSE # bloc repeté pour tous VAR
    ...                 #
CONTINUER_ICI           # après on arrive ici
...
```

L'utilisation du mot clé `in` est différente dans ce contexte qu'avant.

Exemple :

```
somme = 0
for x in [2, 3, 5, 7, 11, 13, 17, 19]:
    print("On ajoute", x)
    somme = somme + x
print("La somme des nombres premiers < 20 est", somme)
```

La boucle `for`

La fonction `range()` retourne un n-uplet des nombres entiers :

- `range(y)` retourne `(0,1,2,..., y-1)`
- `range(x, y)` retourne `(x,x+1,x+2,..., y-1)`
- `range(x, y, s)` retourne `(x,x+s,x+2s,..., x+ns)` avec `x+ns<y` maximal

Application typique de `range()` dans une boucle `for` :

```
for x in range(ITER):
    FAIRE_QUELQUE_CHOSE  # bloc repeté ITER fois
    ...
```

Exemple :

```
print("Les carrés et les cubes des nombres entre 0 et 9:")
for x in range(10):
    print(x**2)
    print(x**3)
    print("\n")
```

La boucle `for`

Avec un str, une boucle `for` se répète pour tous caractères :

```
for caractere in "jeu":
    print(caractere + caractere)    #    "jj
                                    #     ee
                                    #     uu"
```

Boucles `for` imbriquées :

```
animaux = ["Poisson", "Tortue", "Cachalot"]
n = 0
for animal in animaux:
    for caractere in animal:
        if caractere == "o":
            n += 1
print("Le nombre des 'o' dans la liste est", n)
```

Commandes utiles pour les boucles

La commande `break` abandonne une boucle. Exemple :

```
while True:                 # toujours vrai
    i = int(input("Entrer un nombre pair: "))
    if i % 2 == 0:    # vrai si i est pair
        print(i, "/ 2 = ", i / 2)
        break
```

Après une boucle, la commande `else` marque un bloc à exécuter seulement si la boucle n'a pas été abandonnée avec `break` mais s'est terminée régulièrement. Exemple :

```
binaire = input("Entrez un nombre binaire (des 0 et 1): ")
somme = 0
for i in range(len(binaire)): # len(x) = longueur du str x
    bit = int(binaire[i])
    if  bit == 0 or bit == 1:
        somme += bit * 2**(len(binaire) - i - 1)
    else:
        print("Expression non valide")
        break
else:
   print("Ce nombre en notation décimale est", somme)
```

Commandes utiles pour les boucles

Dans une boucle, la commande `continue` saute les instructions restants et continue avec la prochaine itération

```
s = input("Entrer une phrase: ")
for caractere in s:
    if caractere == "e":
        continue  # sauter l'instruction suivante
    print(caractere) # écrire toutes les lettres sauf les 'e'
```

Exemple : Boucles et structures conditionnelles

Un parachutiste est en chute libre pendant 20 s. Après il ouvre son parachute et il descend à une vitesse constante de 2 m/s. On s'intéresse à sa position en fonction du temps.

```
g = 9.81    # accélération gravitationnelle en m/s^2
v = 2.0     # vitesse après ouverture du parachute en m/s

h0 = float(input("Hauteur initiale en m: "))

for t in range(0, 22, 2):     # on affiche h tous les 2 s
    h = h0 - 0.5 * g * t**2   # nouvelle hauteur
    if h <= 0:                # ça fait mal!
        break
    print("A t =", t, "s, la hauteur est de", h, "m.")
else:
    print("Le parachute s'ouvre.")
    while h > 0:
        print("A t = ", t, "s, la hauteur est de", h, "m.")
        t += 10 # on affiche la hauteur tous les 10 s
        h -= 10 * v

print("Atterrissage!")
```

Les structures de contrôle : Erreurs fréquentes

- Deux-points oubliés après `if`, `while`, `for` etc.
- Pour l'indentation des blocs :
  Ne jamais mélanger les éspaces et les tabulatrices.
  Conseillé : eviter les tabulatrices, indentation 4 espaces par niveau
- L'opérateur d'affectation est =, l'opérateur de comparaison est ==
  Donc `a == b` est une expression logique (qui vaut `True` si les valeurs de `a` et `b` sont égales, et `False` sinon) mais `a = b` est une affectation qui attribue à `a` la valeur de `b`.

```
cont = int(input("Combien y a-t-il de continents?"))
if cont = 6:          # Erreur! Ici il faut utiliser ==
    print("C'est correct!")
```

- Boucles infinies : assurez-vous que vos boucles se terminent !

```
x_n, r, i = 0.5, 3.6, 1
while i < 100:
    print(x_n)
    x_n = r * x_n * (1 - x_n)
print("Ca y est!") # Jamais atteint car i ne change pas
```

Les fonctions

Dans ce chapitre

Python

- Les définitions de fonctions
- La commande `return`
- La portée des identifiants
- Les fonctions anonymes et la commande `lambda`
- Les modules et la bibliothèque standard
- Le module `math`

Généralités

- La récursivité
- Les fonctions d'ordre supérieur

Exemples de fonctions

On a déjà employé quelques fonctions intégrées dans Python comme `print()`, `input()` et `range()`. En générale, une fonction est une partie du programme qui

- peut être appellée avec un ou plusieurs paramètre(s) dit argument(s)
- effectue une tâche en fonction de ces arguments
- peut retourner une valeur

Par exemple, comme nous l'avons vu, la fonction `range()` accepte entre 1 et 3 arguments et retourne un n-uplet de nombres entiers.

Autres exemples :

fonction	argument(s)	tâche	valeur de retour
`print()`	un, d'un type quelconque	affichage sur l'écran	aucune
`input()`	un `str`	affiche son argument, attend saisie du clavier	le `str` entré par le clavier
`int()`	un nombre ou `str` qui peut être converti en `int`	convertit son argument en `int`	le résultat de la conversion
`len()`	une séquence	compte le nombre d'élements	le nombre d'élements dans la séquence

Nouvelles fonctions

Voici un exemple d'une définition d'une fonction originale `cube()` :

```
def cube(x):          # un argument, nommé x
    return x ** 3     # retourne x au cube
```

Les instructions dans les définitions de fonction sont exécutées lorsque l'interpreteur tombe sur un appel de fonction :

```
a = cube(5) # appelle la fonction cube() avec l'argument
            #   x=5, affecte la valeur de retour à a
print(a)    # affiche "125"
```

Définir une fonction

La syntaxe pour une définition d'une fonction est

```
def NOM_DE_FONCTION(ARG1, ARG2, ...):
    INSTRUCTION1
    INSTRUCTION2
    ...
```

- Pour les noms des fonctions, les mêmes règles que pour les autres identifiants s'appliquent.
- Une fonction peut accepter un nombre quelconque d'arguments `ARG1`, `ARG2` etc.
- Une fois la fonction définie, on l'appelle avec la commande
  `NOM_DE_FONCTION(VAL1, VAL2, ...)`
  où `VAL1`, `VAL2` etc. sont les valeurs à substituer pour les arguments `ARG1`, `ARG2` etc. pendant cet appel.
  La valeur de l'expression d'appel devient la valeur de retour de la fonction.
- Le bloc suivant la ligne d'en-tête, caractérisé par le mot clé `def`, contient les instructions à exécuter à chaque appel. Comme tous les blocs, il peut contenir des sous-blocs gérés par des structures de contrôle, des appels de fonctions...

Valeurs par défaut des arguments

Il est possible de spécifier des valeurs par défaut pour tous les arguments ou une partie :

```
def NOM_DE_FONCTION(ARG1=DEF1, ARG2=DEF2,...):
    ...
```

Si une valeur par défaut est spécifiée dans la définition d'une fonction, il est facultatif d'inclure la valeur de l'argument correspondant lors d'un appel de fonction.

Exemple : Calculer une approximation de la fonction zêta de Riemann,

$$\zeta(z) = \lim_{N \to \infty} \sum_{k=1}^{N} \frac{1}{k^z}$$

```
def zeta(z, N=100):
  somme = 0.
  for k in range(1, N+1): # k entre 1 et N inclus
    somme += 1/k**z  # ajouter le k-ème terme à la somme
  return somme
```

Possibles appels pour calculer $\zeta(2)$: `zeta(2)` ou `zeta(2, 1000)` ou `zeta(2, N=500)`

Valeurs par défaut des arguments

Si plusieurs arguments ont des valeurs par défaut : Dans un appel, les arguments non nommés doivent toujours précédér les arguments nommés pour éviter toute ambiguïté.

Exemple :

```
# x n'a pas de valeur par défaut. z=1 et y=1 par défaut.
def multiplier(x, y=1, z=1):
    return x * y * z
```

Exemples d'appels de cette fonction :

- `multiplier(25, 5, 2)`
- `multiplier(-2, 16)` (en utilisant la valeur par défaut du dernier argument z)
- `multiplier(3)` (dans ce cas les valeurs par défaut pour z et y sont utilisées)
- `multiplier(17, z=5)` (ce qui pose $x = 17$, $z = 5$, et $y = 1$ sa valeur par défaut)

Par contre, `multiplier(z=5, 17)` est un appel invalide (un argument nommé ne peut pas précéder un argument non nommé)

La commande `return`

La commande `return` peut figurer à un ou plusieurs endroits dans la définition d'une fonction. Dès que l'interpréteur la rencontre, il abandonne la fonction et continue l'exécution du programme à l'endroit de l'appel.
L'argument de `return` est renvoyé et devient la valeur de l'expression d'appel de fonction.

```
def heaviside(r):          # la définition d'une fonction
    if r >= 0:
        return 1.0
    else:
        return 0.0

print("Theta(1) =", heaviside(1))    # un premier appel
print("Theta(-1) =", heaviside(-1)) # un deuxième appel
```

Si une fonction n'est pas terminée par un `return`, ou si `return` est rencontré sans aucun argument, la fonction renvoie l'objet abstrait `None` ("aucun").

Parenthèse : Coding style

Récommandations pour créer du code plus lisible :

- Adopter des conventions cohérents et les suivre partout.
- Ne pas économiser des commentaires.
- Une instruction par ligne. Eviter les point-virgules ou des lignes comme

```
if CONDITION: FAIRE_QC    # légal mais mauvais style
```

- 4 espaces par niveau d'indentation. Pas de tabulatrice.
- Pas d'espace juste après des parenthèses, crochets et accolades ([{
  ni juste avant)]}
  ni juste avant des virgules, point-virgules et deux-points
- un seul espace après , ; :
- un seul espace à chaque coté des opérateurs d'affectation et de comparaison
- Désignations parlantes pour les identifiants importants.
  Préférer des minuscules et éventuellement des chiffres et tirets bas _ pour les variables et fonctions.
- Préférer des majuscules pour les classes en programmation orientée objet.

Deuxième parenthèse : La portée des identifiants

La portée (lexicale) d'un nom de variable est la portion du code où la variable peut être adressée par ce nom. Par défaut, pour les affectations à l'intérieur d'une définition de fonction, la portée des noms des variables n'est que cette définition de fonction. De ce fait, le code

```
def f():
    x = 0    # définir la variable x dans la portée de f()
f()
print(x)     # erreur: x pas défini dans cette portée
```

produit une erreur. Par contre, si une variable est définie hors d'une définition de fonction, on peut tout de même l'utiliser à son intérieur :

```
x = 0
def f():
    print(x)    # variable x définie hors de f()
f()             # mais pas de problème
```

La portée des identifiants

On peut définir une variable, dans la portée locale d'une fonction, avec le le même nom qu'une variable déjà définie hors de la fonction. Cela crée une nouvelle variable, et dans la portée locale, le nom se réfère toujours à cette nouvelle variable locale. Exemple :

```
x = 0       # définit variable globale x
def f():
    x = 1   # définit variable locale x
    print(x)    # "1" (priorité de la variable locale)
f()
print(x)    # "0" (hors portée de la variable locale)
```

Des affectations aux variables globales dans une définition de fonction sont toutefois possibles (mais déconseillées si évitable), avec la commande `global` :

```
x = 0
def f():
    global x # x se référera à la variable globale:
    x = 1    # ceci ne crée pas une nouvelle variable locale
f()
print(x)     # "1"
```

Récursivité

Une fonction peut appeler elle-même :

```
def factorielle(n): # calcule n! recursivement
    if n < 0:       # factorielle pas définie
        print("Erreur: factorielle pas définie.")
        return
    elif n == 0:      # 0! = 1
        return 1
    else:
        return n * factorielle(n - 1)   # n! = n (n-1)!
```

On parle de la récursivité. Attention à la terminaison des algorithmes qui s'en servent !

Fonctions comme arguments

On peut passer des fonctions comme arguments aux autres fonctions :

```
def iterer(f, depart, n_fois):  # f(f(...f(depart)))
    resultat = depart
    for n in range(n_fois):
        resultat = f(resultat)
    return resultat

def logistique(x, r = 3.6):
    return r * x * (1 - x)

print(iterer(logistique, 0.5, 100)) #   0.43172
```

Dans l'appel de `iterer()` le nom de la fonction `f` (`logistique` en ce cas) est traité comme un nom d'une variable.

Fonctions comme valeurs de retour

Une fonction peut renvoyer une autre :

```
def racine(n): # renvoie la fonction "n-ème racine"
    def f(x):
        return x**(1/n)
    return f

print(racine(5)(32))    # affiche "2.0"
```

Une fonction qui soit prend une autre fonction comme argument soit renvoye une fonction est dite une fonction d'ordre supérieur. Les fonctions d'ordre supérieur sont particulièrement importantes dans la programmation fonctionnelle.

Fonctions `lambda`

La commande `lambda` permet des très courtes definitions de fonctions dans une seule ligne. Au lieu de

```
def carre(x):
    return x ** 2
```

on écrit

```
carre = lambda x: x ** 2
```

Plus généralement,

```
lambda ARGUMENTS: EXPRESSION
```

définit une fonction avec arguments `ARGUMENTS` qui retourne `EXPRESSION`.

Fonctions `lambda` : Exemples

Fonctions d'ordre supérieur :

```
def racine(n): # renvoie la fonction "n-ème racine"
    return lambda x: x**(1/n)

print(racine(4)(81))     # affiche "3.0"
```

Listes de fonctions :

```
# une fonction, sa dérivée et sa dérivée seconde
fonctions = [lambda x: 3*x**2 - 2*x,
             lambda x: 6*x - 2,
             lambda x: 6]
```

Les modules

La commande `import` sert à importer du code d'un autre fichier.

Exemple : On enregistre dans un fichier `puissances.py` les définitions de fonction

```
# fichier puissances.py
def carre(x):
    return x**2
def cube(x):
    return x**3
```

On peut ensuite les utiliser dans un autre projet

```
# fichier nouveauprojet.py (dans le même répertoire)
import puissances
print(puissances.carre(42))
```

sans recopier tout. Ou, si on voulait seulement importer la fonction `carre()` :

```
from puissances import carre
print(carre(42))    # ici: pas 'puissances.carre(42)'
```

La bibliothèque standard

Python est fourni avec un grande bibliothèque standard de fonctions pré-definies pour toutes sortes de tâches. Entre autres il y a des modules pour

- manipuler les chaînes de caractères
- manipuler les tableaux de données (NumPy, voir plus tard)
- les fonctions mathématiques (math et cmath, voir ci-dessous, ainsi que NumPy)
- les nombres rationnelles
- accéder et manipuler les fichiers
- les interfaces aux bases de données
- la compression et la sauvegarde des données
- services cryptographiques
- interactions avec le système d'exploitation
- les services de réseau
- les services internet et les pages web
- multimédia
- les interfaces graphiques

La bibliothèque standard : les modules `math` et `cmath`

Fonctions et constantes utiles du module `math` de la bibliothèque standard :

```
import math  # pour importer toutes les fonctionnalités
             #  du module math

# Les constantes e et pi
math.e                  # 2.71828...
math.pi                 # 3.14159...

# Les fonctions trigonométriques
math.sin(1.2)
math.cos(math.pi)
math.tan(0)

# Les fonctions trigonométriques inverses
math.asin(1/2)
math.acos(0.5)
math.atan(-1)
```

La bibliothèque standard : les modules `math` et `cmath`

Fonctions et constantes utiles du module `math` de la bibliothèque standard :

```
import math

# Les fonctions exponentielle et logarithme
math.exp(-3.0)
math.log(1.0)      # logarithme naturel
math.log(4, 2)     # logarithme de 4 de base 2

# La racine carrée
math.sqrt(2.0)          # équivalent: 2**(1/2)
```

- Toutes ces fonctions prennent des arguments du type `float` (ou des arguments `int`, que Python convertit automatiquement en `float`).
- Si on veut les appliquer aux nombres complexes, il faut importer le module `cmath` au lieu de `math`.

Quelques autres modules de la bibliothèque standard

Le module `time` met à disposition des fonctions pour accéder à l'horloge interne de l'ordinateur. Exemples :

```
import time

# Retourner un str représentant la date et l'horaire
# présente
time.asctime()

# Retourner un float qui représente le nombre de secondes
# depuis 1 janvier 1970 0:00:00 UTC
time.time()

# Arrêter l'exécution du programme pendant 7.5 secondes
time.sleep(7.5)
```

Quelques autres modules de la bibliothèque standard

Le module `random` contient des fonctions pour générer des nombres (pseudo-)aléatoires. Exemples :

```
import random

# Retourner un nombre pseudoaléatoire entre 0 et 1
# avec une distribution uniforme
random.random()

# Retourner un élément aléatoire d'une liste
L = [1, 19, 23, 47]
random.choice(L)

# Retourner un entier aléatoire entre a (inclu)
# et b (exclu) avec une distribution uniforme
a, b = 10, 20
random.randint(a, b) # un entier aléatoire entre 10 et 19
```

Les fonctions : Erreurs fréquentes

- Deux-points oubliés après la commande def
- Il faut qu'une fonction soit définie avant l'appel.

```
x, y = ma_fonction() # erreur: ma_fonction
                     # pas encore definie ici!

def ma_fonction():   # définition trop tardive
    a = int(input("Entrez un nombre entier:"))
    return a // 5, a % 5
```

- Pour les fonctions importées :
  attention à la différence entre import et from ... import :

```
from math import sqrt
x = sqrt(10)          # pas math.sqrt()
import cmath
y = cmath.exp(1.0J) # pas exp()
```

- Une définition de fonction sans appel ne forme pas un programme complet !