INTRODUCTION AU LANGAGE PYTHON

C. Schlick

schlick@u-bordeaux.fr

CHAPITRE 1

Concepts & Vocabulaire

Informatique / Informatics

► Ensemble des activités humaines liées au traitement de l'information par un ordinateur

Information / Information

► Toute donnée, réelle ou abstraite, manipulable par la pensée humaine

- **Origine** : Claude Shannon, 1948
  « A Mathematical Theory of Communications »
- **Théorème** : *Toute information est représentable par une suite (éventuellement infinie) de 0 et de 1*

Traitement de l'information / Info processing

► Transformation quelconque effectuée entre une information en entrée et une information en sortie

- *En corollaire du théorème de Shannon, il s'agit donc simplement de transformer une séquence de 0 ou 1 en une autre séquence de 0 ou 1*

Ordinateur / Computer

► Dispositif matériel qui fonctionne comme un ***automate déterministe à états finis*** (**ADEF**) dont les transitions sont programmables

Concepts & Vocabulaire

ADEF / Deterministic Finite-State Machine

- Modèle mathématique d'une machine possédant plusieurs états et dont les changements d'états (appelés ***transitions***) s'effectuent de manière déterministe, en réaction à des signaux externes
- Ordinateur = Concrétisation matérielle d'un ADEF en y ajoutant la programmabilité des transitions
- Ordinateur = Deux composantes indispensables
  - Composante ***matérielle***
  - Composante ***logicielle***

Concepts & Vocabulaire

Matériel / Hardware

- Carte mère : *processeur, mémoire, horloge*
- Périphériques : *dispositifs d'interaction, dispositifs de stockage, dispositifs de connexion…*

Logiciel / Software

- Description des actions à réaliser pour obtenir une transformation entre l'information disponible en entrée, et l'information souhaitée en sortie
- Code logiciel **<=>** Programmation de l'automate

Concepts & Vocabulaire

Programmation / Programming

- Art de la conception et de la réalisation de code logiciel ⟶ ***littérature, rhétorique***

ou

- Science de la conception et de la réalisation de code logiciel ⟶ ***logique, algorithmique***

ou

- Ingénierie de la conception et de la réalisation de code logiciel ⟶ ***génie industriel, productique***

Concepts & Vocabulaire

Programmation <=> Littérature ?

- Ecrire un texte de qualité dans une langue particulière (langage de programmation)

Programmation <=> Logique ?

- Définir l'enchaînement des états de l'automate entre un point de départ et un point d'arrivée

Programmation <=> Génie industriel ?

- Réaliser l'ensemble des étapes partant d'un cahier des charges pour arriver à un produit fini

Concepts & Vocabulaire

Conséquence : La maîtrise de la programmation nécessite des compétences variées et complexes

⟶ **Métier à part entière** ⟵

- Programmeur (années 1960-1980)
- Analyste-programmeur (années 1980-1995)
- Développeur logiciel (depuis 1995)

  ***Spécialisations** : développeur système, développeur web, développeur multi-média, développeur jeux, développeur middleware...*

Concepts & Vocabulaire

Trois principales familles de logiciels

- **Utilitaires** = traitement d'information liée à l'informatique : *configuration système, gestion de fichiers, archivage de données…*
- **Applicatifs** = traitement d'information extérieure à l'informatique : *bureautique, navigation réseau, messagerie, calcul scientifique, CAO, FAO…*
- **Ludo-éducatifs** = visualisation d'information (sans traitement par l'utilisateur) : *jeux vidéos, simulateurs, lecteur de données multi-média, aide à l'apprentissage...*

Concepts & Vocabulaire

Structure d'un logiciel

*Décomposition **systématique** en deux entités*

Noyau / Kernel

- Définition des ***actions à réaliser*** et organisation des ***données à manipuler*** par l'automate, afin d'obtenir l'information souhaitée à l'arrivée

Interface utilisateur / User Interface

- Définition et organisation de la ***communication bidirectionnelle*** (entrée/sortie) entre l'automate et l'utilisateur du logiciel

Concepts & Vocabulaire

Algorithmique & Structures de Données

⟶ *cf. UE ASD1 et UE ASD2*

Interface ⊂ Interfaces Homme / Machine

- Interface (en mode) texte (~ 1950)

  ***Command Line Interface** (CLI)*

- Interface (en mode) graphique (~ 1980)

  ***Graphical User Interface** (GUI)*

Concepts & Vocabulaire

Programmation / Programming

- Conception et réalisation de code logiciel

Code logiciel / Software code

- Ensemble des instructions destinées à l'automate
- Chaque code logiciel existe sous deux formes :
  - **Code source** = texte écrit par le développeur s'exprimant dans un ***langage de programmation***
  - **Code exécutable** = traduction du code source en une séquence de 0 ou 1 (***langage machine***) compréhensible par l'automate (≈ processeur)

Concepts & Vocabulaire

Lang. Programmation ⟶ Lang. Machine

Trois modes de traduction possibles :

- **Interprétation :** Chaque phrase en LP est convertie individuellement en une phrase en LM
  - Avantage : *Portabilité maximale*
  - Inconvénient : *Efficacité très faible*
- **Compilation :** L'ensemble du texte en LP est converti en un texte complet en LM
  - Avantage : *Efficacité maximale*
  - Inconvénient : *Portabilité très faible*

Concepts & Vocabulaire

Lang. Programmation ⟶ Lang. Machine

Trois modes de traduction possibles :

- **Hybride :** L'ensemble du texte en LP est converti en LMV (langage machine pour un automate virtuel, indépendant de l'architecture matérielle) puis ce texte est converti, phrase par phrase, en LM au moment de l'exécution sur l'ordinateur
  - Avantage : *Portabilité maximale*
  - Inconvénient : *Efficacité quasi optimale*

La plupart des langages récents sont hybrides

Concepts & Vocabulaire

Comme une langue humaine, un langage de programmation est caractérisé par 2 éléments :

- **Grammaire :** correspond à la syntaxe du langage, c'est-à-dire la manière d'agencer les mots pour en faire des phrases compréhensibles
- **Vocabulaire :** correspond à l'ensemble des mots ayant une sémantique spécifique pour le langage
  - Partie rigide: *mots-clés, opérateurs, délimiteurs*
  - Partie flexible: *valeurs littérales, identificateurs*

Comment décrire une syntaxe ?

Décrire un langage dans un autre langage ?

- Les langues humaines ne sont pas adaptées
- Utilisation d'une notation spécifique, basée sur des opérateurs algébriques (≈ équations)

Plusieurs systèmes de notation existent

- EBNF (Extended Backus-Nauer Form)
  Notation la plus complète et la plus utilisée, mais (relativement) complexe
- UNIX "**`usage`**" notation (issu du système UNIX)
  Certaines limitations mais très simple à maîtriser

UNIX "usage" notation

Notation basée sur 6 symboles spécifiques

= | ... ' *literal* ' < *symbol* > [*symbol*]

Quelques exemples pour comprendre :

- binary-digit = < '0' | '1' >
- binary-digits = <binary-digit> [binary-digit …]
- digit = < '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' >
- digits = <digit> [digit …]
- integer = ['+' | '−'] <digits>
- float = <integer> < '.' > [digits] ['E' | 'e' integer]
- float = ['+' | '−'] < '.' > <digits> ['E' | 'e' integer]

Introduction au langage Python

CHAPITRE 2

Description générale du langage Python

Structure du code Python

- **Code Python =** Ensemble de paquetages
  - Concrètement : *paquetage = dossier (disque)*
  - Symboliquement : *paquetage <=> livre*
- **Paquetage / Package =** Ensemble de modules
  - Concrètement : *module = fichier (disque)*
  - Symboliquement : *module <=> chapitre*
- **Module / Module =** Ensemble de blocs
  - Concrètement : *bloc = séquence de lignes*
  - Symboliquement : *bloc <=> paragraphe*

Structure du code Python

- **Bloc / Block =** Ensemble d'instructions
  - Conc. : *instruction = séquence de caractères*
  - Symb. : *instruction <=> phrase grammaticale*
- **Instruction / Statement =** Entité élémentaire du langage, obtenue en combinant cinq types de symboles spécifiques :
  - Symboles rigides : ***mots réservés / keywords, délimiteurs / delimiters, opérateurs / operators***
  - Symboles flexibles : ***littéraux / literals, identificateurs / identifiers***

Mots réservés / Keywords

Six familles de mots réservés : *chacune reliée à une catégorie spécifique d'instructions*

1. **Affectation :** `=  +=  -=  *=  /=  ...`
2. **Définition :** `class/def/lambda/global/nonlocal/[local]/with ... as/from ... import ... as`
3. **Répétition :** `while/for`
4. **Condition :** `if/elif/else/assert/try/except`
5. **Rupture :** `break/continue/return/yield/raise`
6. **Inaction :** `pass`

Délimiteurs / Delimiters

Huit familles de délimiteurs :

1. **Bloc :** `<head>':' <indented body> '\n'`
2. **Instruction :** `<statement> <';'|'\n'>`
3. **Commentaire :** `'#' <comment> '\n'`
4. **Séquence :** `<expression','> [expression',' ...]`
5. **Tuple :** `'(' <sequence> ')'`
6. **Liste :** `'[' <sequence> ']'`
7. **Ensemble & Dictionnaire :** `'{' <sequence> '}'`
8. **Chaîne :** `'...' | "..." | '''...''' | """..."""`

Opérateurs / Operators

Neuf familles d'opérateurs (par priorité) :

1. **Groupes :** `(...)  [...]  {...}`
2. **Attributs :** `x(y...)  x[y...]  x.y`
3. **Exposant :** `**`
4. **Opérateurs unaires :** `-x  +x  ~x`
5. **Opé. multiplicatifs :** `*  /  //  %  @`
6. **Opé. additifs :** `+  -`
7. **Opé. binaires :** `<<  >>  :  &  :  ^  :  |`
8. **Comparaisons :** `==  !=  <  <=  >  >=  is  is␣not  in  not␣in`
9. **Opé. booléens :** `not  :  and  :  or`

Expression ≠ Instruction

Instruction = Action à effectuer

- *Chaque instruction figurant dans le code va modifier l'état courant de l'automate (transition)*
  - Symb. : *instruction <=> phrase grammaticale*

Expression = Donnée à construire

- *Chaque expression figurant dans le code va rajouter une nouvelle donnée pour l'automate*
- *Une expression est une combinaison algébrique mêlant opérateurs, délimiteurs, valeurs littérales*
  - Symb. : *expression <=> groupe nominal*

Donnée = Type + Valeur

Donnée = Séquence binaire + Codage

- *Toute donnée manipulée par l'automate est constituée d'une séquence binaire (0 ou 1) de taille variable, complétée par la description du codage utilisé*

Codage <=> Type

- *Les données de même nature sont codées de manière identique, ce codage est appelé "* **Type***" de la donnée*

Séquence binaire <=> Valeur

- *Une fois le codage spécifié, la séquence binaire associée à une donnée est définie de manière unique, et est appelée "* **Valeur***" de la donnée*

Donnée = 2 axes de classification

Donnée mutable ou non ?

- *Encoder la séquence binaire associée à une donnée est une opération souvent complexe et coûteuse*
- *Lorsque le développeur sait qu'une donnée ne va pas être modifiée durant le déroulement du programme, il est préférable de le spécifier au moment de la création*
- *Tout langage de programmation permet de différencier les données* ***constantes*** *et les données* ***variables****.*
- *Mais ces notions ont souvent un sens assez différent d'un langage à l'autre. Il est donc préférable d'utiliser les termes* ***mutable*** *pour les données modifiables et* ***non-mutable*** *pour les données figées.*

Donnée = 2 axes de classification

Donnée itérable ou non ?

- *Certains langages de programmation différencient les types* ***simples*** *(qui stockent une valeur unique) et les types* ***complexes*** *(qui stockent des valeurs multiples)*
- *Mais là encore, ces notions ont souvent un sens assez différent d'un langage à l'autre, et la distinction simple vs. complexe est donc sujet à controverse*
- *Python propose une terminologie qui n'est pas basée sur le nombre de valeurs stockées, mais sur le fait que ces valeurs puissent (ou non) être parcourues à l'aide d'une boucle. On va donc utiliser les termes* ***itérable*** *ou* ***non-itérable*** *pour caractériser ces deux cas*

CHAPITRE 3

Catégories d'instructions disponibles en Python

Catégories d'instructions

Python définit 6 catégories d'instructions

- *Chaque instruction correspond à un mot-clé spécifique*

1. **Affectation :** `=  +=  -=  *=  /=  ...`
2. **Définition :** `[local]/nonlocal/global/def/class/with ... as/from ... import ... as`
3. **Répétition :** `while/for`
4. **Condition :** `if/elif/else/assert/try/except`
5. **Rupture :** `break/continue/return/yield/raise`
6. **Inaction :** `pass`

Instructions de définition

- Un **identificateur** est un symbole arbitraire créé pour nommer les diverses entités définies dans le code : *module, classe, fonction, donnée…*
- Cette association entre identificateur et entité est créée via une **instruction de définition**
- Deux contraintes sur les identificateurs :
  - Un identificateur est une suite quelconque de caractères pris parmi les 63 caractères suivants

    `a ... z  A ... Z  0 ... 9  _`
  - Le 1er caractère ne peut pas être un chiffre pour les différencier des symboles numériques

Instructions de définition

- Les **espaces de noms** sont des dictionnaires qui associent à chaque identificateur, la zone mémoire où est stockée l'entité associée, ainsi que le codage binaire utilisé pour cette entité
- Ces espaces sont définis hiérarchiquement :
  - au niveau de chaque fonction (***local level***)
  - au niveau de chaque objet (***object level***)
  - au niveau de chaque classe (***class level***)
  - au niveau de chaque module (***global level***)
  - au niveau de l'interpréteur (***builtin level***)

Instructions de définition

Affectation : `=`

- *L'instruction d'affectation permet d'associer un identificateur avec une **donnée** (valeur littérale, expression algébrique...)*
- *Syntaxe à utiliser :*

```
name = data
```

- *L'interprétation intuitive de cette opération est de de se dire qu'on colle une étiquette sur une donnée*

Exemple d'utilisation

```
# affectation simple
a = 0
# affectation imbriquée
b = c = 1
# affectation multiple
d, e, f = 0, 1, 2
# modification
a = 2*b + 3*c # a = 5
c += 1 # <=> c = c + 1
b /= 2 # <=> b = b / 2
# permutation
a, b = b, a
d, e, f = e, f, d
```


Instructions conditionnelles

Condition : `if`

- *Une instruction conditionnelle permet de forcer la vérification d'un prédicat avant d'exécuter un bloc*
- *Syntaxe à utiliser :*

```
if <test>:
   <block>
[elif <test>:
   <block> ...]
[else:
   <block>]
```

Exemple d'utilisation

```
# condition simple
if x < 0: y = -1
# alternative simple
if x < 0: y = -1
else: y = 1
# alternative multiple
if x < -10: y = -2
elif x > 10: y = 2
elif x < 0: y = -1
elif x > 0: y = 1
else: y = 0
```


Instructions de répétition

Boucle : `while`

- *Une boucle permet de d'exécuter un bloc tant qu'un prédicat est valide*
- *Syntaxe à utiliser :*

```
while <test>:
   <block>
[else:
   <block>]
```

- *Le bloc lié à* `else` *est exécuté après la boucle, sauf s'il y a eu rupture avec l'instruction* `break`

Exemple d'utilisation

```
# boucle sans rupture
a, b = 1, 0
while a <= 5:
  a, b = a+1, a+b
else:
  print('a =',a,'b =',b)
# --> a = 6  b = 15

# rupture avec break
a, b = 1, 0
while a <= 5:
  if a == 4: break
  a, b = a+1, a+b
else:
  print('a =',a,'b =',b)
# --> n'affiche rien
```


Instructions de répétition

Boucle : `while`

- *Une boucle permet de d'exécuter un bloc tant qu'un prédicat est valide*
- *Syntaxe à utiliser :*

```
while <test>:
   <block>
[else:
   <block>]
```

- *Le bloc lié à* `else` *est exécuté après la boucle, sauf s'il y a eu rupture avec l'instruction* `break`

Exemple d'utilisation

```
# rupture avec continue
a, b = 1, 0
while a <= 5:
  if a == 4: continue
  a, b = a+1, a+b
print('a =',a,'b =',b)
# --> boucle infinie

# rupture avec continue
a, b = 0, 0
while a <= 5:
  a = a+1
  if a == 4: continue
  b = a+b
print('a =',a,'b =',b)
# --> a = 6  b = 11
```

Instructions de répétition

Itération : `for`

- *Une itération permet de d'exécuter un bloc pour chaque donnée présente dans une séquence*
- *Syntaxe à utiliser :*

```
for <var> in <seq>:
  <block>
[else:
  <block>]
```

- *Même remarque que précédemment pour le bloc lié à* **`else`**

Exemple d'utilisation

```
# itération sans rupture
b = 0
for a in (1,2,3,4,5):
  b += a
else:
  print('a =',a,'b =',b)
# --> a = 5  b = 15

# rupture avec break
b = 0
for a in (1,2,3,4,5):
  if a == 4: break
  b += a
else:
  print('a =',a,'b =',b)
# --> n'affiche rien
```


Instructions de répétition

Itération : `for`

- *Une itération permet de d'exécuter un bloc pour chaque donnée présente dans une séquence*
- *Syntaxe à utiliser :*

```
for <var> in <seq>:
  <block>
[else:
  <block>]
```

- *Même remarque que précédemment pour le bloc lié à* **`else`**

Exemple d'utilisation

```
# rupture avec break
b = 0
for a in (1,2,3,4,5):
  if a == 4: break
  b += a
print('a =',a,'b =',b)
# --> a = 5  b = 6

# rupture avec continue
b = 0
for a in (1,2,3,4,5):
  if a == 4: continue
  b += a
else:
  print('a =',a,'b =',b)
# --> a = 5  b = 11
```


Instructions de définition

Fonction : `def`

- *La définition de fonction permet d'associer un identificateur avec un* ***bloc d'instructions***
- *Syntaxe à utiliser :*

```
def name(…):
  block
```

- *Une fonction peut utiliser des données en entrée (appelées* ***arguments****) et renvoyer des données en sortie (appelées* ***retours****)*

Exemple d'utilisation

```
# 0 argument, 0 retour
def f():
  print('aaa bbb')
# 2 arguments, 0 retour
def g(a, b):
  print(a, b)
# 0 argument, 3 retours
def f():
  return -1, 0, 1
# 2 arguments, 2 retours
def g(a, b):
  return a+b, a-b
```


Instructions de définition

Fonction : `def`

- *Après la définition d'une fonction, les instructions du bloc associé peuvent être exécutées par un* ***appel de fonction***
- *Une fonction renvoie toujours une donnée en retour à l'instruction qui a effectué l'appel. Lorsqu'il n'y a pas pas de retour explicite, c'est la valeur* **`None`** *qui est renvoyée*

Exemple d'utilisation

```
def f(x):
  print(x)
def g(x):
  return x, x+1, x+2

# appels de fonctions
f(0) # --> 0 (affichage)
a = f(1) # a = None
g(0) # retour perdu
a = g(0) # a = (0,1,2)
a, b, c = g(1)
# a = 1, b = 2, c = 3
a, b = g(0) # ValueError
f(g(0)) # --> (0,1,2)
```

Instructions de définition

Fonction : `def`

- *Les arguments d'une fonction peuvent avoir des valeurs par défaut, en utilisant la syntaxe :*

  `(arg=value, …)`

- *Attention, si on combine des arguments standards avec des arguments qui possèdent une valeur par défaut, ceux-ci doivent impérativement se placer à droite de la séquence*

Exemple d'utilisation

```
def f(a, b=1, c=2):
  return a, b, c
def g(a=0, b=1, c=2):
  return a, b, c
# appels de fonctions
x = f(1,2,3) # x = (1,2,3)
x = f(1,2) # x = (1,2,2)
x = f(1) # x = (1,1,2)
x = f() # TypeError
y = g(1,2,3) # y = (1,2,3)
y = g(1,2) # y = (1,2,2)
y = g(1) # y = (1,1,2)
y = g() # y = (0,1,2)
```


Instructions de définition

Générateur : `def`

- *Un générateur peut être vu comme une **fonction itérable** qui retourne des valeurs à la demande*
- *Un générateur se définit exactement comme une fonction, sauf qu'il utilise* **`yield`** *et non* **`return`** *pour la valeur de retour*
- *La fonction* ***`range`*** *est de très loin, le générateur le plus commun en Python*

Exemple d'utilisation

```
def f(): # fonction
  return 0 ; return 1
  return 2 ; return 3
def g(): # générateur
  yield 0 ; yield 1
  yield 2 ; yield 3
x = (f(),f(),f(),f())
# x = (0,0,0,0)
y = g()
# y = <generator object>
y = tuple(g())
# y = (0,1,2,3)
z = tuple(range(4))
# z = (0,1,2,3)
```


Instructions de définition

Générateur : `def`

- *A chaque appel, un générateur redémarre à partir de la dernière instruction* ***`yield`*** *exécutée lors de l'appel précédent*
- *L'appel d'un générateur doit donc nécessairement se faire au sein d'une itération : soit une boucle, soit une conversion vers une donnée itérable (par exp.* **`tuple`**, **`list`**, **`set`***)*

Exemple d'utilisation

```
def squares(p=0,q=-1):
  while p != q:
    yield p**2
    p += 1
# mise en oeuvre
a = list(squares(2,7))
# a = [4,9,16,25,36]
for p in squares():
  print(p, end=' ')
# --> 0 1 4 9 16 25 36
49 64 81 100 121 144 …
# boucle infinie !
```


Introduction au langage Python

CHAPITRE 4

Types de données standards en Python

Types de données standards

Python fournit une large variété de types

- *Les types prédéfinis de Python ne correspondent pas à des mots-clés du langage, mais à des* **classes**, *au sens de la programmation par objets (cf. UE Prog 2)*

[i] = itérable, [o] = ordonné, [m] = mutable

- Vide : **`None`**
- Booléen : **`bool`**
- Entier : **`int`**
- Réel : **`float`**
- Complexe : **`complex`**
- Relation : **`lambda`**
- Tranche : **`slice`**
- Séquence : **`tuple`** *[i,o]*
- Liste : **`list`** *[i,o,m]*
- Chaîne unicode : **`str`** *[i,o]*
- Ensemble : **`set`** *[i,m]*
- Groupe : **`frozenset`** *[i]*
- Dictionnaire : **`dict`** *[i,m]*
- Intervalle : **`range`** *[i,o]*

Type `None` et Type `bool`

Vide : `None`

- *Ensemble ne contenant qu'une seule valeur (mot réservé) :* **`None`**

Booléen : `bool`

- *Ensemble ne contenant que deux valeurs (mots réservés) :* **`True,False`**
- *Opérateurs possibles :*
  `not or and`
  `== !=`
- *Conversion :* **`bool()`**

Exemple d'utilisation

```
>>> False, not False
(False, True)
>>> True, not True
(True, False)
>>> None, not None
(None, True)
>>> not not None
False
>>> bool(None)
False
>>> bool(0), bool(1)
(False, True)
```


Type `None` et Type `bool`

Vide : `None`

- *Ensemble ne contenant qu'une seule valeur (mot réservé) :* **`None`**

Booléen : `bool`

- *Ensemble ne contenant que deux valeurs (mots réservés) :* **`True,False`**
- *Opérateurs possibles :*
  `not or and`
  `== !=`
- *Conversion :* **`bool()`**

Exemple d'utilisation

```
>>> 0 == False, 1 == True
(True, True)
>>> 2 == False, 2 == True
(False, False)
>>> False or True,
    False and True
(True, False)
>>> False or False,
    False and False
(False, False)
>>> True or True,
    True and True
(True, True)
```


Type `int`

Entier : `int`

- *Codage des éléments de l'ensemble* $\mathbb{Z}$
- *Les valeurs numériques sont représentables en décimal (sans préfixe), binaire (préfixe 0B ou 0b) octal (préfixe 0O ou 0o), hexadécimal (0X ou 0x)*
- *Conversion inverse avec les fonctions standards :*
  `bin() oct() hex()`

Exemple d'utilisation

```
>>> 1, 11, 111
(1, 11, 111)
>>> -1, -0, +0, +1
(-1, 0, 0, 1)
>>> 0B1, 0B11, 0B111
(1, 3, 7)
>>> 0O1, 0O11, 0O111
(1, 9, 73)
>>> 0X1, 0X11, 0X111
(1, 17, 273)
>>> bin(7), hex(273)
('0B111', '0X111')
```

Type `int`

Entier : `int`

- *Opérateurs possibles :*

```
+    -    *    /
//   %    **
<<   >>   ~
&    |    ^
is   is not
==   !=   >
>=   <=   <
not  and  or
```

- *Fonctions prédéfinies :*

```
abs  divmod
```

Exemple d'utilisation

```
>>> 1 + 2*3 - 2**3
-1
>>> 7 / 4, 7 // 4, 7 % 4
(1.75, 1, 3)
>>> 7 << 4, 7 >> 4, ~7
(112, 0, -8)
>>> 7 & 4, 7 | 4, 7 ^ 4
(4, 7, 3)
>>> 7 == 0B111, 11 == 0XB
(True, True)
>>> 1 < 4 < 7, 1 < 4 > 7
(True, False)
```


Type `int`

Entier : `int`

- *Opérateurs possibles :*

```
+    -    *    /
//   %    **
<<   >>   ~
&    |    ^
is   is not
==   !=   >
>=   <=   <
not  and  or
```

- *Fonctions prédéfinies :*

```
abs  divmod
```

Exemple d'utilisation

```
>>> abs(1), abs(-1)
(1, 1)
>>> divmod(7,3)
(2, 1) # <=> 7//3, 7%3
>>> 1234**56
129911902554871451941032084396235137754657820101273923843790127046242594330550946489256784853624729020106139515647384910944921186523865849056275359066262352911682504769929216
# précision infinie
```


Type `float`

Réel : `float`

- *Codage des éléments de l'ensemble* $\mathbb{R}$
- *Les valeurs numériques sont définies en notation décimale ou scientifique (suffixe* `E` *ou* `e`*)*
- *Précision contrôlable par la fonction* `round()`
- *Conversion entier / réel par les fonctions :* `int()` `float()`

Exemple d'utilisation

```
>>> 1., .1, -1., -.1
(1.0, 0.1, -1.0, -0.1)
>>> 2/3  # 16 chiffres
0.6666666666666666
>>> round(2/3, 4)
0.6667
>>> round(0.5000000001)
1  # entier le + proche
>>> 2E3, 12.34E-5
(2000.0, 0.0001234)
>>> int(0.99), float(1)
(0, 1.0)
```


Type `float`

Réel : `float`

- *Opérateurs possibles :*

```
+    -    *    /
//   %    **
is   is not
==   !=   >
>=   <=   <
not  and  or
```

- *Fonctions mathématiques usuelles (valeurs dans* $\mathbb{R}$ *)*

```
from math import *
```

Exemple d'utilisation

```
>>> abs(-1.2 ** 3.4)
1.858729691979481
>>> 1.0 is 1, 1.0 == 1
(False, True)
>>> from math import *
acos acosh asin asinh
atan atan2 atanh ceil
cos cosh degrees e
erf erfc exp factorial
floor gamma gcd hypot
isclose isfinite isinf
isnan log log10 log2
pi pow radians sin
sinh sqrt tan tanh
```

Type `complex`

Complexe : `complex`

- *Codage des éléments de l'ensemble* $\mathbb{C}$
- *Suffixe* `J` *ou* `j` *pour la partie imaginaire*
- *Accès à la partie réelle et à la partie imaginaire :* `c.real  c.imag`
- *Conversion* `bool`, `int` *ou* `float` *vers* `complex`

Impossible dans l'autre sens car dimensions incompatibles !

Exemple d'utilisation

```
>>> 0j, 1j, 2+3j, 1j**2
(0j, 1j, 2+3j, -1+0j)
>>> c = (1+2j)/4
>>> c, c.real, c.imag
(0.25+0.5j, 0.25, 0.5)
>>> c.conjugate()
(0.25-0.5j)
>>> complex(1.0, 2.3)
1+2.3j
>>> float(1+0j)
TypeError: can't convert
complex to float
```


Type `complex`

Complexe : `complex`

- *Opérateurs possibles :*

```
+   -   *   /   **
==  !=  is  is not
not   and   or
```

Rappel: $\mathbb{C}$ *est non ordonné*

- *Fonctions mathématiques usuelles (valeurs dans* $\mathbb{C}$ *)*

```
from cmath import *
```

Exemple d'utilisation

```
>>> abs(3+4j) # module
5.0
>>> from cmath import *
acos acosh asin asinh
atan atanh cos cosh e
exp isclose isfinite
isinf isnan log log10
phase pi polar rect sin
sinh sqrt tan tanh
>>> phase(3+4j) # arg
0.927295218001612
>>> polar(3+4j)
(5.0,0.927295218001612)
```


Type `tuple`

Séquence : `tuple`

- *Série ordonnée, itérable, non-mutable de valeurs arbitraires, séparées par des virgules (a.k.a.* ***CSV****)*
- *On peut délimiter par des ( ) pour plus de lisibilité*
- *Lorsque la série est vide, les ( ) sont obligatoires*
- *Lorsque la série est un singleton, une virgule finale est obligatoire*

Exemple d'utilisation

```
>>> 1,2.3,4+5j,None
(1, 2.3, 4+5j, None)
>>> (1,2.3,4+5j,None)
(1, 2.3, 4+5j, None)
>>> () # séquence vide
()
>>> 1, # singleton
(1,)
>>> (1,) # singleton
(1,)
>>> (1) # entier !!!
1
```


Type `tuple`

Séquence : `tuple`

- *Opérateurs possibles :*

```
+   *   ==   !=   is
is not not and or
```

- *La création de séquences multidimensionnelles est possible simplement par imbrication de séquences*
- *Fonctions prédéfinies :*

```
min   max   sum
len   all   any
```

Exemple d'utilisation

```
>>> (1, 2, 3) + (4, 5)
(1, 2, 3, 4, 5)
>>> (1, 2) * 3
(1, 2, 1, 2, 1, 2)
>>> ((1, 2, 3), (4, 5))
((1, 2, 3), (4, 5))
>>> a = (1,2,3)
>>> b = (4,5)
>>> a + b
(1, 2, 3, 4, 5)
>>> (a, b)
((1, 2, 3), (4, 5))
```

Type `tuple`

Séquence : `tuple`

- *Opérateurs possibles :*

```
+   *   ==   !=   is
is not not and or
```

- *La création de séquences multidimensionnelles est possible simplement par imbrication de séquences*
- *Fonctions prédéfinies :*

```
min   max   sum
len   all   any
```

Exemple d'utilisation

```
>>> a = (0, 1, 2)
>>> b = (1, 2, 3)
>>> len(a), len(a+b)
(3, 6)
>>> min(0), max(a+b)
(1, 3)
>>> sum(a), sum(a+b)
(3, 9)
>>> any(a), any(b)
(True, True)
>>> all(a), all(b)
(False, True)
```


Type `tuple`

Séquence : `tuple`

- *Les éléments (**items**) d'une séquence sont accessibles en spécifiant un indice entre crochets*
- *Un indice positif ou nul est compté depuis le début de la séquence, alors qu'un indice négatif est compté depuis la fin*
- *Les éléments individuels ne sont pas modifiables, puisqu'une séquence est toujours non-mutable*

Exemple d'utilisation

```
>>> a = (0,1,4,9,16,25)
>>> a[0], a[2], a[5]
(0, 4, 25)
>>> a[-1], a[-4], a[-6]
(25, 4, 0)
>>> a[9]
IndexError: tuple index out of range
>>> a[0] = 1
TypeError: tuple object does not support item assignment
```


Type `tuple`

Séquence : `tuple`

- *Les éléments d'un **tuple** sont aussi accessibles par tranche (**slice**) à l'aide d'un triple indice :*

`[start]<:>[stop][:step]`

- **`start`** = *indice de début (valeur 0, par défaut)*
- **`stop`** = *indice de fin (nb. d'items, par défaut)*
- **`step`** = *longueur du pas (valeur 1, par défaut)*

Exemple d'utilisation

```
>>> a = (0,1,4,9,16,25)
>>> a[:]
(0,1,4,9,16,25)
>>> a[2:4], a[1:-1]
((4,9), (1,4,9,16))
>>> a[:3], a[3:]
((0,1,4), (9,16,25))
>>> a[4:2], a[2:9]
((), (4,9,16,25))
>>> a[0:5:2], a[5::-4]
((0,4,16), (25,1))
```


Type `list`

Liste : `list`

- *Structure similaire à une séquence, sauf qu'une liste est **mutable***
- *Les valeurs d'une liste sont obligatoirement délimitées par des [ ]*
- *La création de listes multidimensionnelles est possible (comme pour les séquences) simplement par imbrication de listes*

Exemple d'utilisation

```
>>> [1,2.3,4+5j,None]
[1, 2.3, 4+5j, None]
>>> [] # liste vide
[]
>>> [1] # singleton
[1]
>>> [[1, 2, 3], [4, 5]]
[[1, 2, 3], [4, 5]]
>>> tuple([1,2,3])
(1, 2, 3)
>>> list((1,2,3))
[1, 2, 3]
```

Type `list`

Liste : `list`

- *L'accès (par indice ou par tranche) aux éléments d'une liste utilise la même notation par crochets que pour les séquences*
- *Mais comme une liste est mutable, cette même notation par crochets est également utilisable pour modifier, supprimer ou insérer des éléments*

Exemple d'utilisation

```
>>> a = [0,1,4,9,16,25]
>>> a[0], a[2], a[-1]
(0, 4, 25)
>>> a[9]
IndexError: list index out of range
>>> a[2:-1], a[5::-4]
([4, 9, 16], [25, 1])
>>> a[2] = 2
>>> a[-2] = 12
>>> a
[0, 1, 2, 9, 12, 25]
```


Type `list`

Liste : `list`

- *L'accès (par indice ou par tranche) aux éléments d'une liste utilise la même notation par crochets que pour les séquences*
- *Mais comme une liste est mutable, cette même notation par crochets est également utilisable pour modifier, supprimer ou insérer des éléments*

Exemple d'utilisation

```
>>> a[3:] = [5,6]
>>> a
[0, 1, 2, 5, 6]
>>> a[3:3] = [3,4]
>>> a
[0, 1, 2, 3, 4, 5, 6]
>>> a[::3] = [0,0,0]
>>> a
[0, 1, 2, 0, 4, 5, 0]
>>> a[-3:-1] = []
>>> a[1:3] = []
>>> a
[0, 0, 0]
```


Type `list`

Liste : `list`

- *Les listes disposent de fonctionnalités internes (appelées **méthodes**), accessibles avec la notation par point :*

  `list.method(arguments)`

- *Quelques méthodes :*

`index`	`count`
`clear`	`copy`
`append`	`extend`
`sort`	`reverse`
`pop`	

Exemple d'utilisation

```
>>> a = [0,1,2,1,1,3,0]
>>> a.index(3)
5
>>> a.count(1)
3
>>> a.reverse()
>>> a      # ≠ a[::-1]
[0, 3, 1, 1, 2, 1, 0]
>>> a.sort()
>>> a
[0, 0, 1, 1, 1, 2, 3]
>>> a.clear()
>>> a
[]
```


Type `list`

Liste : `list`

- *Les listes disposent de fonctionnalités internes (appelées **méthodes**), accessibles avec la notation par point :*

  `list.method(arguments)`

- *Quelques méthodes :*

`index`	`count`
`clear`	`copy`
`append`	`extend`
`sort`	`reverse`
`pop`	

Exemple d'utilisation

```
>>> a.append(1)
>>> a
[1]
>>> a.extend([2,3,4])
>>> a
[1, 2, 3, 4]
>>> a.append([5,6,7])
>>> a
[1, 2, 3, 4, [5, 6, 7]]
>>> a.pop(), a.pop(2)
([5, 6, 7], 3)
>>> a
[1, 2, 4]
```

Type `str`

Chaîne Unicode : `str`

- *Une chaîne Unicode est une séquence où chaque élément est un caractère au standard Unicode*
- *Chaînes courtes (ligne unique) délimitées par :*

  `'...'` *ou* `"..."`

- *Chaînes longues (multi-lignes) délimitées par :*

`'''...'''` *ou* `"""..."""`

Exemple d'utilisation

```
>>> 'aaa "bbb" ccc'
'aaa "bbb" ccc'
>>> "aaa 'bbb' ccc"
"aaa 'bbb' ccc"
>>> '' # chaîne vide
''
>>> s = """
aaa bbb
ccc ' ddd
eee " fff"""
>>> s
'\naaa bbb\nccc \'
ddd\n eee \" fff'
```


Type `str`

Chaîne Unicode : `str`

- *Caractères spéciaux :*

  `\\` `\n` `\t`
  `\'` `\"` `\uXXXX`

- *Opérateurs possibles :*

  `+` `*` `==` `!=` `is`
  `is not not and or`

- *Fonctions prédéfinies :*

  `len` `ord` `chr`
  `min` `max`

Exemple d'utilisation

```
>>> "a\tb\nc\td"
'a\tb\nc\td'
>>> print('a\tb\nc\td')
a       b
c       d
>>>'\u25C4\u2665\
u25BA''◄♥►'
>>> ('a b' + 'cd ') * 3
'a bcd a bcd a bcd '
>>> ord('!'), ord('■')
(33, 9632)
>>> chr(33),chr(0X2665)
('!', '♥')
```


Type `str`

Chaîne Unicode : `str`

- *L'accès (par indice ou par tranche) aux caractères d'une chaîne se fait avec la notation par crochets*
- *En tant que structure non mutable, les caractères d'une chaîne ne sont pas modifiables directement.*
- *Pour modifier une chaîne il faut obligatoirement en recréer une nouvelle à partir de la première*

Exemple d'utilisation

```
>>> s = "abracadabra"
>>> s[0], s[2], s[-3]
('a', 'r', 'b')
>>> s[3:-5], s[::-1]
('aca','arbadacarba')
>>> s[-4:] = 'obro'
TypeError: str does not
support item assignment
>>> s = s[:-4] + 'obro'
>>> s
'abracadobro'
```


Type `str`

Chaîne Unicode : `str`

- *Python fournit un grand nombre de méthodes de manipulation de chaînes*
- *Quelques méthodes :*

`find`	`rfind`
`count`	`center`
`ljust`	`rjust`
`lower`	`upper`
`title`	`capitalize`
`endswith`	`startswith`
`split`	`rsplit`
`join`	`strip`

Exemple d'utilisation

```
>>> s = "abracadabra"
>>> s.find('bra')
1
>>> s.rfind('a',2,-2)
7
>>> 'OO'.center(18)
'        OO        '
>>> 'OO'.center(18,'-')
'--------OO--------'
>>> t = 'aaa bbb ccc'
>>> t.capitalize()
'Aaa bbb ccc'
>>> t.title()
'Aaa Bbb Ccc'
```

Type `str`

Chaîne Unicode : `str`

- *Python fournit un grand nombre de méthodes de manipulation de chaînes*
- *Quelques méthodes :*

`find`	`rfind`
`count`	`center`
`ljust`	`rjust`
`lower`	`upper`
`title`	`capitalize`
`endswith`	`startswith`
`split`	`rsplit`
`join`	`strip`

Exemple d'utilisation

```
>>> s = "abracadabra"
>>> s.replace('a','o')
'obrocodobro'
>>> s.split('r')
['ab', 'acadab', 'a']
>>> s.rsplit('b',1)
['abracada', 'ra']
>>> t = 'aa bb  cc  dd'
>>> t.split()
['aa','bb','cc','dd']
>>> ':'.join(t.split())
'aa:bb:cc:dd'
```


Type `range`

Intervalle : `range`

- *Générateur de valeurs entières définies à l'aide de trois paramètres :*

`[start,]<stop>[,step]`

- `start` = *indice de début (valeur 0, par défaut)*
- `stop` = *indice de fin (obligatoire)*
- `step` = *longueur du pas (valeur 1, par défaut)*

Exemple d'utilisation

```
>>> range(7)
range(0,7) # générateur
>>> list(range(7))
[0, 1, 2, 3, 4, 5, 6]
>>> [*range(7)]
[0, 1, 2, 3, 4, 5, 6]
>>> tuple(range(7))
(0, 1, 2, 3, 4, 5, 6)
>>> *range(7),
(0, 1, 2, 3, 4, 5, 6)
>>> [*range(2,8)]
[2, 3, 4, 5, 6, 7]
>>> [*range(5,26,5)]
[5, 10, 15, 20, 25]
```


Type `range`

Intervalle : `range`

- *Les valeurs de l'intervalle ne sont générées qu'à la demande (**générateur**) :*
  - *soit lors d'une conversion en séquence ou en liste*
  - *soit lors d'un parcours de l'intervalle par itération en utilisant une boucle* `for`

Exemple d'utilisation

```
>>> (p%2 for p in
     range(7))
 <generator object>
>>> [p%2 for p in
     range(7)]
 [0, 1, 0, 1, 0, 1, 0]
>>> [(p, p*p) for p
     in range(1,7,2)]
 [(1,1), (3,9), (5,25)]
>>> for p in range(7):
      print(p, end=' ')
 0 1 2 3 4 5 6
```


Type `slice`

Tranche : `slice`

- *Structure très similaire à un intervalle, mais dont l'utilisation est réservée à la définition d'une tranche pour une donnée itérable*
- *Différences de* `slice` *par rapport à* `range` *:*
  - *les indices négatifs sont comptés à partir de la fin de la donnée itérable*
  - *un indice* `None` *fournit les valeurs par défaut*

Exemple d'utilisation

```
>>> a = [0,1,4,9,16,25]
>>> b = slice(None)
>>> a[b] # <=> a[:]
[0, 1, 4, 9, 16, 25]
>>> b = slice(4)
>>> a[b] # <=> a[:4]
[0, 1, 4, 9]
>>> b = slice(-2,None)
>>> a[b] # <=> a[-2:]
[16, 25]
>>> b = slice(1,None,3)
>>> a[b] # <=> a[1::3]
[1, 16]
```

Type `set`

Ensemble : `set`

- *Ensemble non-ordonné de valeurs arbitraires sans éléments dupliqués*
- *Les valeurs d'un ensemble sont obligatoirement délimitées par des { }*
- *Ensemble non-ordonné :*
  - *l'ordre des valeurs n'est pas toujours respecté*
  - *l'accès aux éléments par indice est impossible*

Exemple d'utilisation

```
>>> {1,2.3,4+5j,None}
{2.3, 1, (4+5j), None}
>>> set() # ensemble vide
set()
>>> {1} # singleton
{1}
>>> {5,2,1,5,2,-1,5,0}
{0, 1, 2, 5, -1}
>>> a = {1,2.3,4+5j}
>>> a[0]
TypeError: 'set' object
does not support indexing
```


Type `set`

Ensemble : `set`

- *Opérateurs possibles :*

```
|     &     -     ^
==    !=    >
>=    <=    <
in    not in
```

- *Fonctions prédéfinies :*

```
min   max   sum
len   all   any
```

- *Conversions entre les structures itérables :*

```
tuple  list  set
```

Exemple d'utilisation

```
>>> a = {0, 1, 2}
>>> b = {1, 2, 3}
>>> a | b, a - b
({0, 1, 2, 3}, {0})
>>> a & b, a ^ b
({1, 2}, {0, 3})
>>> 3 in a, (a & b) < a
(False, True)
>>> len(b),sum(b),all(b)
(3, 6, True)
>>> tuple(a)
(0, 1, 2)
```


Type `set`

Ensemble : `set`

- *Comme pour les autres structures, Python fournit plusieurs méthodes pour manipuler les données dans les ensembles*
- *Quelques méthodes :*

```
add     update
pop     discard
clear   copy
```

Exemple d'utilisation

```
>>> a.clear()
>>> a.add(1)
{1}
>>> a.update([2,3,4])
{1, 2, 3, 4}
>>> a.pop()
1
>>> a
{2, 3, 4}
>>> a.discard(3)
{2, 4}
>>> a.discard(5)
{2, 4}
```


Type `frozenset`

Groupe : `frozenset`

- *Structure similaire à un ensemble, sauf qu'elle est **non-mutable***
- *Pas de délimiteur attitré, il faut impérativement utiliser le constructeur* `frozenset(items)` *où* `items` *doit être une structure itérable*
- *Les opérateurs d'un groupe sont les mêmes que ceux d'un ensemble*

Exemple d'utilisation

```
>>> s = {1,2.3,4.5}
>>> frozenset(s)
frozenset({2.3,1,4.5})
>>> frozenset()
    # groupe vide
frozenset()
>>> frozenset({1})
    # singleton
frozenset({1})
>>> frozenset(range(3))
frozenset({0, 1, 2})
```

Type dict

Dictionnaire : dict

- *Association entre un ensemble de clés et un ensemble de valeurs*
- *Les clés d'un dictionnaire sont obligatoirement des données non mutables*
- *Les valeurs peuvent être des données mutables*
- *Un dictionnaire est défini en créant une séquence de couples* **key:value** *délimitée par des {}*

Exemple d'utilisation

```
>>> {1:'aaa',2.3:[4,5]}
{2.3: [4, 5], 1: 'aaa'}
>>> {'A':1,'B':2,'C':3}
{'C': 3, 'A': 1, 'B': 2}
>>> {} # dict vide
{}
>>> {(1,2):3, (4,5):6}
{(1, 2): 3, (4, 5): 6}
>>> {[1,2]:3, [4,5]:6}
TypeError: unhashable
  type: 'list'
# les clés ne doivent
pas être mutables
```


Type dict

Dictionnaire : dict

- *Lorsque que les clés sont des chaînes Unicode, on peut également utiliser la syntaxe* **name = value** *combinée avec l'appel au constructeur* **dict**
- *Une dernière alternative pour créer un dictionnaire est de combiner 2 listes et/ou séquences à l'aide de la fonction standard* **zip** *avant l'appel à* **dict**

Exemple d'utilisation

```
>>> dict(aaa=1, bbb=2)
{'aaa': 1, 'bbb': 2}
>>> dict(A=1, B=2, C=3)
{'C': 3, 'A': 1, 'B': 2}
>>> a = [1,2,3]
>>> b = 'ABC'
>>> c =('aaa','bbb')
>>> dict(zip(a,b))
{1: 'A', 2: 'B', 3: 'C'}
>>> dict(zip(b,a))
{'C': 3, 'A': 1, 'B': 2}
>>> dict(zip(c,a))
{'aaa': 1, 'bbb': 2}
```


Type dict

Dictionnaire : dict

- *Opérateurs possibles :*
  == != in not in
- *Fonctions prédéfinies :*
  min max sum
  len all any
  (sur les clés uniquement)
- *L'accès aux valeurs d'un dictionnaire (en lecture et en écriture) se fait à partir de la clé, en utilisant la notation par crochets*
- *Pas d'accès par tranche*

Exemple d'utilisation

```
>>> d = {'A':1, 'B':2}
>>> 1 in d, 'A' in d
(False, True)
>>> len(d),max(d),all(d)
(2, 'B', True)
>>> d['A']
1
>>> d['C'] = 3
>>> d['B'] = 5
>>> del d['A']
>>> d
{'C': 3, 'B': 5}
```


Type dict

Dictionnaire : dict

- *Comme pour les autres structures, Python fournit plusieurs méthodes pour manipuler les données dans les dictionnaires*
- *Quelques méthodes :*

clear	**keys**
values	**items**
get	**update**
pop	**popitem**

Exemple d'utilisation

```
>>> d = {'A':1, 'B':2}
>>> d.keys(), d.values()
(['A', 'B'], [1, 2])
>>> d.items()
[('A', 1), ('B', 2)]
>>> d.get('D', 0)
0
>>> d.update(C=3, D=4)
>>> d.pop('B')
2
>>> d
{'C': 3, 'A': 1, 'D': 4}
```