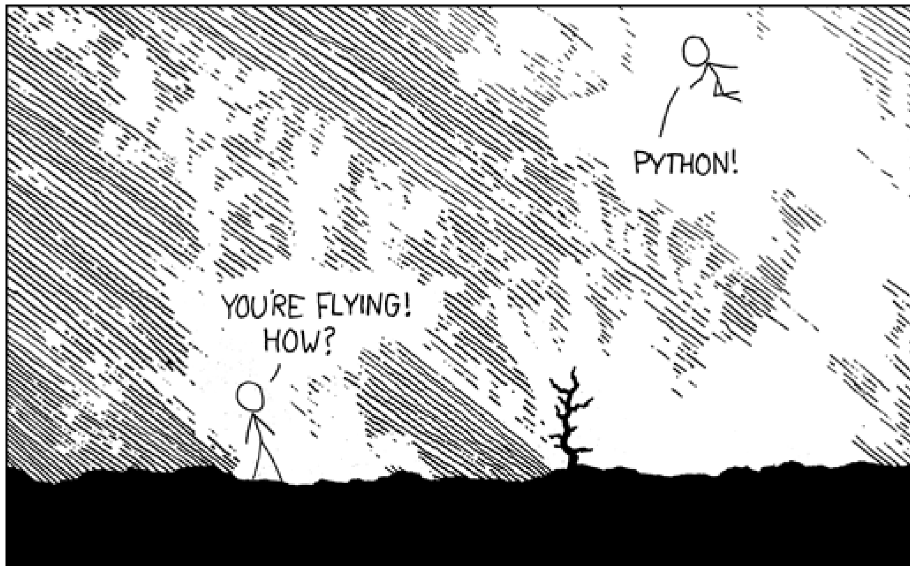
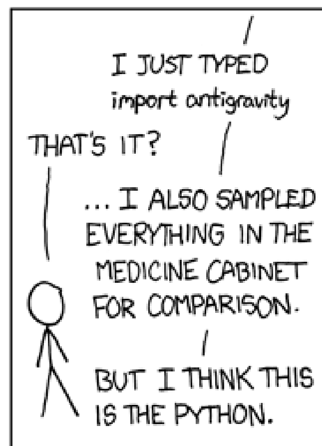
Aide-mémoire Python pour l'ITC

Pierre Karpman
Lycée Champollion

2024-09





https://xkcd.com/353/

§ Objectifs

Cet aide-mémoire a pour but de présenter de façon à peu près exhaustive tous les aspects du langage Python (version $\geq 3.12.4$) au programme de l'*informatique tronc commun* des CPGE scientifiques, mentionnés à l'Annexe A du programme.

§ Préliminaires : environnement de travail

Cette description suppose que vous utilisez un « terminal en ligne de commande », et est à adapter si ce n'est pas le cas.

¶ **Interpréteur interactif.** L'environnement standard Python fournit un interpréteur interactif permettant de définir et exécuter du code Python. Cet interpréteur peut s'accéder depuis un terminal en faisant simplement :

```
> python
```

On quitte l'interpréteur en utilisant la combinaison de touche `Ctrl + D` ou en appelant la fonction `exit` (avec ou sans argument).

Il existe d'autres interpréteurs interactifs, par exemple `ipython`, que l'on conseille vivement d'utiliser à la place de l'interpréteur standard (si disponible sur votre machine).

On supposera par défaut l'utilisation d'un tel interpréteur interactif pour exécuter notre code, mais ce n'est pas la seule possibilité.

¶ **Fichier + éditeur = bonheur.** On conseille *très vivement* de n'utiliser l'interpréteur interactif *que* pour appeler des fonctions définies par ailleurs, ou éventuellement pour tester des instructions ou expressions élémentaires. Votre travail (par exemple les fonctions que vous devez écrire dans le cadre d'un TP) doit se trouver dans un fichier d'extension `.py` (par exemple `src.py`), que vous éditerez *via* un éditeur de texte adapté.

¶ **Accéder aux fichiers depuis l'interpréteur.** Vous pouvez facilement accéder aux fonctions définies dans un fichier `src.py` depuis un interpréteur en lançant celui-ci depuis le répertoire contenant `src.py` et faisant :

```
import src # alternative : from src import *
```

Puis pour par exemple exécuter la fonction `fun(a)` définie dans `src.py`, faire :

```
src.fun(a) # alternative : fun(a)
```

¶ **Prendre en compte les changements dans un fichier.** Si vous avez importé `src.py` dans un interpréteur et édité (et sauvegardé) `src.py`, les changements ne seront pas immédiatement visibles depuis l'interpréteur. Vous pouvez alors quitter celui-ci et tout recommencer, ou faire :

```
from importlib import reload
from sys import modules
reload(modules["src"])
```

 © Pierre Karpman

§ Expressions arithmétiques et booléennes.

Le langage Python définit des *expressions* dont les *évaluations* permettent d'effectuer des calculs divers. Ces expressions peuvent être de plusieurs types (cf. ci-dessous) et se construisent inductivement à partir de *littéraux* (par ex. `1`, `1.0`, `True`...), de *variables* (par ex. `a`), et d'*opérateurs* dont les opérandes sont elles-mêmes des expressions. Dans ce dernier cas, l'utilisation de parenthèses permet d'indiquer l'ordre dans lequel les sous-expressions sont évaluées ; en leur absence, cet ordre est donné par les règles de priorité entre opérateurs (par ex. : la multiplication est prioritaire sur l'addition). La plupart des appels de fonction produisent également une valeur, et donc une expression.

¶ **Expressions entières.** Soit `ae1`, `ae2` deux expressions python de type `int`, on peut notamment construire les expressions entières suivantes, *via* les opérateurs correspondant :

— Addition : `ae1 + ae2`
— Soustraction : `ae1 - ae2`
— Produit : `ae1 * ae2`
— Quotient de la division entière : `ae1 // ae2` (seul le cas d'opérandes positives est au programme ; dans ce cas, le quotient est toujours associé à un reste positif, cf. ci-dessous). Peut être la cause d'une `ZeroDivisionError`.
— Reste de la division entière : `ae1 % ae2` (seul le cas d'opérandes positives est au programme ; dans ce cas, le reste est toujours positif). Peut être la cause d'une `ZeroDivisionError`.
— Exponentiation : `ae1 ** ae2`

Exemples :

— `2 * 3`
— `2 * (3 + a)`
— `2 * 3 + a`

¶ **Expressions flottantes.** Soit `fe1`, `fe2` deux expressions python de type `float`, on peut notamment construire les expressions flottantes suivantes :

— Addition : `fe1 + fe2`
— Soustraction : `fe1 - fe2`
— Produit : `fe1 * fe2`
— Division « réelle » : `fe1 / fe2`
— Exponentiation : `fe1 ** fe2`

¶ **Expressions booléennes.** Soit `be1`, `be2` deux expressions python de type `bool`, on peut notamment construire les expressions booléennes suivantes :

— Et logique : `be1 and be2`
— Ou logique : `be1 or be2`
— Négation : `not be1`

Soit `fabe1` et `fabe2` deux expressions python de type `int`, `float` ou `bool` (pas nécessairement identiques), on peut notamment construire les expressions booléennes suivantes :

— Test d'égalité `fabe1 == fabe2`
— Test de différence `fabe1 != fabe2`
— Test d'inégalité `fabe1 <= fabe2` ; `fabe1 >= fabe2`
— Test d'inégalité stricte `fabe1 < fabe2` ; `fabe1 > fabe2`

 © Pierre Karpman

Les opérateurs `and` et `or` sont *paresseux* : ils n'évaluent pas la seconde opérande (l'opérande à droite) si le résultat de l'expression peut déjà être déterminé par la seule évaluation de la première (l'opérande à gauche). Ceci peut être mis à profit dans certains cas ; par exemple, l'expression `(d != 0) and (a % d == 0)` n'entraîne jamais de `ZeroDivisionError`, car si `d` est nul la première opérande s'évalue à `False` et le résultat (`False`) de l'évaluation du `and` est maintenant connu : la seconde opérande ne sera donc jamais évaluée. Ce comportement est aussi important si `be1` ou `be2` sont des expressions ayant des *effets de bord*, c'est à dire qu'en plus de s'évaluer à une valeur booléenne elles modifient l'*état* du programme. Dans ce cas, cette modification aura lieu ou non en fonction de la valeur de la première opérande ; par exemple, l'expression `(x < 7) or (L.pop() == 2)` peut entraîner ou non une modification de la liste `L` en fonction de la valeur de `x`.

§ Instructions & structures de contrôle

Python définit un nombre limité de structures de contrôle qui permettent de définir un *flot* d'exécution, et ultimement donnent sa puissance expressive au langage de programmation.

De même que les expressions se définissent inductivement à partir de littéraux, de variables, et d'opérateurs construisant une expression à partir de sous-expressions, les instructions sont définies inductivement à partir d'instructions de base (faisant généralement intervenir des expressions) et de structures de contrôle permettant d'enchaîner des instructions.

¶ **Affectation de variable.** Cette instruction permet de définir des variables et de leur affecter une valeur, donnée par l'évaluation d'une expression. Une variable existante peut être modifiée « absolument » (par une expression qui ne fait pas référence à la variable elle-même) ou « relativement ».

— `a = 3` : déclaration de la variable `a` et affectation de la valeur `3`, ou modification « absolue » d'une variable `a` préexistante (pas de différence faite en Python...)
— `a = a + 1` : modification « relative » de la variable `a` : après exécution (avec succès) de l'instruction, la valeur de `a` sera la valeur précédant l'exécution, augmentée de `1`

Il existe un certain nombre de *sucres syntaxiques* couplant une affectation et une opération. Ceux-ci ne sont cependant **pas au programme** :

— `a += b` : équivalent à `a = a + b`
— `a *= b` : équivalent à `a = a * b`

Etc.

¶ **Instructions conditionnelle.** Ces instructions permettent d'exécuter une ou plusieurs instructions en fonction du résultat d'une ou plusieurs expressions booléennes. Les *blocs* des instructions exécutées dans les différents cas sont délimités *via* l'indentation, cf. ci-dessous.

```
if be1: # be1 : expression booléenne
    # instructions si be1 vaut True
elif be2: # optionnel
    # instructions si be1 vaut False
    # et be2 vaut True
elif be3: # optionnel
    # instructions si be1, be2 valent False
    # et be2 vaut True
else: # optionnel
```

 © Pierre Karpman

```
    # instructions si be1, be2, be3 valent False
# instructions exécutées en sortie de conditions
```

¶ **Boucle non bornée.** Cette instruction permet d'exécuter « en boucle » (d'*itérer*) une ou plusieurs instructions en fonction du résultat d'une expression booléenne (dont la valeur peut (et généralement doit) varier en fonction des traitements effectués dans le *corps* de la boucle).

```
while be:
   # instructions du corps de boucle exécutées
   # tant que l'expression booléenne be
   # vaut True
# instructions exécutées en sortie de boucle
```

Il existe aussi une instruction `break` permettant d'interrompre prématurément l'exécution d'une boucle. En cas de *boucles imbriquées* (une boucle contenant une ou plusieurs autres boucles dans son corps), on ne sort que de la boucle ayant exécuté le `break`, et non pas de *toutes* les boucles.

Il existe également certains sucres syntaxiques **fort possiblement hors programme** qui permettent d'utiliser `while` avec des expressions non booléennes : par ex., `while L:` avec `L` une liste permet d'itérer le corps de boucle tant que `L` est non-vide : c'est équivalent à `while len(L) > 0:`.

¶ **Boucle bornée** Cette instruction permet d'itérer une ou plusieurs instructions un certain nombre de fois connu à l'avance. Cette instruction est très *expressive* en Python. De façon générale, elle consiste en :

```
for x in I:
    # corps de boucle permettant
    # de traiter itérativement (et un à un)
    # tout élément x de l'objet itérable I
```

Les instructions du corps de boucle ont accès aux éléments de l'objet `I` et varient (a priori) à chaque itération.

Les « objets itérables » sont très nombreuses en Python ; celles au programme sont :

— `range(a)` : les entiers de 0 à $a-1$ (inclus) : la variable `x` prendra successivement les valeurs 0, 1, etc.
— `range(a, b)` : les entiers de a à $b-1$ (inclus)
— `range(a, b, c)` : les entiers de a à $b-1$ (pas nécessairement atteint) par incrément de c, possiblement négatif (dans ce cas, la plus petite valeur pouvant être atteinte devient $b+1$)
— `L` : une liste
— `T` : un tuple
— `D` : les clefs d'un dictionnaire
— `D.keys()` : pareil, explicitement
— `D.items()` : les couples *(clef, valeur)* d'un dictionnaire
— `D.values()` : les valeurs d'un dictionnaire **(curieusement hors programme)**
— `s` : les caractères d'une chaîne de caractères

Il est aussi possible de sortir prématurément d'une boucle `for` *via* un `break`, mais c'est **possiblement hors programme**.

 © Pierre Karpman

§ Fonctions

Les fonctions jouent un rôle essentiel pour structurer le développement. Notamment, le « découpage » d'une tâche en de multiples fonctions doit être réfléchi, et trop ou trop peu de fonctions nuit à la lisibilité du code (et par là, à l'efficacité du développement, de la relecture etc.)

¶ **Définition.** De façon minimale, une fonction se définit comme :

```
def fun(arg1, arg2):
    # ...
    # corps de la fonction dans lequel
    # arg1 et arg2 sont accessibles comme variables
    # ...
    return res # renvoie un résultat
```

Le nombre d'argument d'une fonction est quelconque (et possiblement égal à zéro) mais fixe (dans le cadre du programme). S'il est attendu que la fonction renvoie un résultat (ce qui n'est pas nécessaire), son corps doit contenir au moins une instruction `return res` avec `res` une expression. Dans le cas contraire la fonction termine lorsqu'il n'y a plus d'instructions à exécuter. Même dans ce cas, il est possible d'utiliser une instruction `return` non suivie d'une expression. Une fonction peut contenir plus d'une instruction `return`, et celles-ci peuvent être à l'« intérieur » de structures de contrôle `if`, `while` etc. ; l'effet est le même dans tous les cas : la fonction termine son exécution et renvoie la valeur de l'éventuelle expression associée.

¶ **Portée des variables** Dans le corps d'une fonction, les variables accessibles sont les arguments de la fonction, les variables *locales* définies dans la fonction, ainsi qu'éventuellement les variables *globales* définies dans le contexte appelant la fonction. Sauf cas particulier (peu probable dans le cadre de la CPGE), c'est une *très mauvaise idée* de référencer une variable globale depuis une fonction.

Au cas où un argument ou une variable locale à la fonction a le même nom qu'une variable globale au contexte appelant, la première *occulte* la seconde dans le corps de la fonction : c'est elle qui sera évaluée ou éventuellement modifiée. Par exemple :

```
def fun1(a):
    a = 1
def fun2():
    a = a + 1
```

donnent lieu à l'interaction suivante dans un interpréteur :

```
In :  a = 2
In :  fun1(a)
In :  a
Out:  2
In :  fun2():
UnboundLocalError: cannot access local variable 'a' where it is not
associated with a value
```

 © Pierre Karpman

¶ **Politique d'appel.** L'appel d'une fonction se fait simplement par `fun(a, b)`. Python suit une *politique d'appel* par *valeur* : les expressions données en argument à la fonction sont d'abord évaluées, et la fonction est ensuite appelée sur le résultat de cette évaluation, qui est affecté dans le corps de la fonction à une variable locale du nom de l'argument. Ce même fonctionnement général peut néanmoins avoir des effets concrètement différents en fonction du type des arguments, pour la même raison que la *copie* d'une variable se comporte différemment en fonction de son type, cf. ci-dessous.

§ Types structurés

En plus des types de base déjà mentionnés (`bool`, `int`, `float`), Python fournit de nombreux types *structurés* qui permettent de « combiner » (inductivement) des objets de types de base (ou structurés).

¶ **Tuples.** Un *tuple* est une suite finie d'éléments. En Python, la structure `tuple` correspondant à cette notion est *immuable* : une fois un tuple construit, il n'est pas possible de modifier ses éléments. C'est une différence importante avec le type `list` ci-dessous.

Les tuples Python peuvent contenir des éléments de types différents, produisant dans tous les cas un élément de type `tuple` (c'est une différence notable par rapport à OCaml, par exemple).

Construction de tuples :

```
t = () # tuple vide
t = 1, True # construction d'un tuple
t = (1, True) # variante
t = (1, (True, False)) # les éléments peuvent eux-mêmes être structurés
t = tup1 + tup2 # concaténation de tuples
t = tup1 * i # concaténation répétée (i fois) d'un tuple
t = tup1[a:b] # tuple correspondant aux éléments d'indice a (inclus) à b
              # (exclus) du tuple tup1. Les indices commencent à zéro
t = tuple(a) # HORS PROGRAMME. Construction d'un tuple contenant les éléments
             # de a (par exemple une liste)
```

Lecture des éléments d'un tuple :

```
l = len(t) # nombre d'éléments dans un tuple
x = t[i] # i : indice entre 0 et len(t) - 1. Possible erreur IndexError
a, b = t # équivalent à a = t[0] ; b = t[1], à condition que t contienne
         # *exactement* deux (ici) éléments. Possible erreur ValueError
```

Un tuple Python étant immuable, ses éléments ne peuvent (normalement) pas être modifiés. Lors de la construction d'un tuple, les éventuelles variables sont évaluées et le tuple est construit « par valeur ». Attention cependant aux tuples contenant des éléments de types structurés non immuables, typiquement des listes :

```
In :  t = 1, 2
In :  t[0] = 0
TypeError: 'tuple' object does not support item assignment
In :  L = [1, 2]
In :  t = 0, L
In :  L[0] = 0
In :  t
```

 © Pierre Karpman

```
Out:  (0, [0, 2])
In :  t = 0, L.copy()
In :  L[0] = 1
In :  t
Out:  (0, [0, 2])
```

¶ **Chaînes de caractères.** Les chaînes de caractère (ou *string*) Python peuvent être assimilées à des tuples dont les éléments sont tous de même type (qui est lui-même « chaîne de caractère »). À ce titre, elles sont immuables et supportent les mêmes opérations de concaténation, répétition, extraction de tranche, calcul de la longueur ou accès par indice que n'importe quel tuple. Une chaîne de caractère peut aussi être directement construite comme :

```
s = '' # chaîne vide
s = 'toto' # variante : s = "toto"
s = 'toto\n' # \n : caractère spécial « retour à la ligne »
```

L'intérêt premier des chaînes de caractères est qu'elles peuvent être lues ou écrites depuis ou vers un fichier ou une sortie, notamment via la fonction `print` (cf. ci-dessous).

La bibliothèque standard du langage Python définit de très nombreuses fonctions pour manipuler les chaînes de caractère. La seule d'entre-elles au programme est la fonction (techniquement, méthode) `split` qui étant donnée une chaîne de caractère renvoie (par défaut) la liste des chaînes séparées par des espaces la composant :

```
In :  s = 'il fait beau et chaud'
In :  s.split()
Out:  ['il', 'fait', 'beau', 'et', 'chaud']
In :  W = 'pif paf pouf'.split() # on peut appeler split directement
                                 # sur un littéral
In :  for w in W:
          print(w)
pif
paf
pouf
```

¶ **Listes.** Comme les tuples, les listes Python sont des suites finies d'éléments de types divers. Une différence majeure est qu'une liste n'est *pas immuable* (elle est *mutable*) : il est à la fois possible de modifier la valeur (et même le type !) d'un élément d'une liste existante, et d'ajouter ou supprimer des éléments. Les listes supportent les mêmes opérations de concaténation, répétition, extraction de tranche, calcul de la longueur ou accès pas indice que les tuples. Elles supportent aussi des opérations supplémentaires en lien avec leur caractère non immuable :

```
L = [] # liste vide
L = [1, 2, True] # construction directe
L[0] = 2 # modification d'un élément
L.append(False) # ajout d'un élément en dernière position
                # (ici, à l'indice 3, après True)
x = L.pop() # double effet : renvoie une expression de valeur
            # du dernier élément (ici False) et supprime cet
```

 © Pierre Karpman

```
            # élément de L
            # possible erreur IndexError
```

Les listes Python peuvent aussi être construites *via* une syntaxe à la fois élégante et pratique : soit I un objet itérable (cf. ci-dessus), une liste contenant tous les éléments de I peut se construire comme :

```
L = [x for x in I]
```

Ceci est tout simplement équivalent à :

```
L = []
for x in I:
    L.append(x)
```

De façon générale, les éléments ajoutés à la liste peuvent être des expressions faisant intervenir la variable d'itération. Par exemple :

```
[2*x for x in range(10)] # les entiers pairs de 0 à 18
```

Enfin, on peut rendre l'ajout des éléments conditionnel, en ajoutant un test. Ceci est malheureusement **hors programme** :

```
[2*x for x in range(10) if x % 3 == 0] # [0, 6, 12, 18]
```

Un aspect *délicat* des listes Python (partagé avec les dictionnaires, cf. ci-dessous) est la façon dont se comportent les copies (et passage d'argument de fonction). Le programme d'informatique de tronc commun ne permet pas d'expliquer dans le détail la raison des différences avec les autres types vus en CPGE, et on se contentera de résumer les principaux effets observables. En résumé, par défaut, les affectations de liste ne font que créer des *alias* qui *référencent* la *même* liste.

```
In :  L = [1, 2]
In :  G = L # G n'est qu'un « alias » pour L
In :  L[0] = 0
In :  G
Out:  [0, 2] # G « a été modifiée comme L »
In :  mypop(L) # fonction d'unique instruction L.pop()
In :  L
Out:  [0] # L a été modifiée par la fonction
```

Ceci mène notamment à un piège classique dont l'exemple ci-dessous est une instance. Supposons qu'on souhaite créer une liste « à deux dimensions », c'est à dire une liste contenant deux listes. On pourrait pour cela faire :

```
In :  L = [[1]*2]*2
In :  L
Out:  [[1, 1], [1, 1]]
In :  L[0][0]
Out:  1
```


© Pierre Karpman

ce qui a l'air de fonctionner. Mais :

```
In :  L[0][0] = 0
In :  L
Out:  [[0, 1], [0, 1]]
```

Les deux listes « internes » sont des alias l'une de l'autre, rendant cette construction peu utile !

Il existe toutefois une fonction (techniquement, une *méthode*) `copy` qui permet de créer une « vraie » copie (néanmoins *superficielle*) d'une liste : la liste renvoyée n'est plus un simple alias mais une nouvelle liste contenant (initialement) les mêmes valeurs que son argument. Cependant, les éléments de la liste ne *sont pas* eux-mêmes copiés avec `copy`, et peuvent donc eux-même être de simples alias :

```
In :  L = [1, 2]
In :  G = L.copy()
In :  L[0] = 0
In :  L
Out:  [0, 2]
In :  G
Out:  [1, 2] # G est réellement une autre liste
In :  L = [[1, 2], [3, 4]]
In :  G = L.copy()
In :  L[0][0] = 0
In :  L
Out:  [[0, 2], [3, 4]]
In :  G
Out:  [[0, 2], [3, 4]] # semble n'avoir rien changé ici
In :  L[0] = [1, 2]
In :  L
Out:  [[1, 2], [3, 4]]
In :  G
Out:  [[0, 2], [3, 4]] # G[0] n'a pas été changé
```

REMARQUE : La création d'un alias d'une liste ne « coûte pas cher » : le coût est essentiellement identique à la copie d'une variable d'un type de base. C'est un avantage par rapport à la copie, qui de façon évidente doit parcourir tous les éléments de la liste et a donc un coût linéaire en sa longueur.

¶ **Dictionnaires.** Les dictionnaires Python sont des ensembles mutables de *valeurs* de types quelconques indexées par des *clefs* de type « hashable » (parmi les types au programme, cela concerne tous les types sauf les listes et les dictionnaires) :

```
D = {} # dictionnaire vide
D = {3:4, "a":5} # dictionnaire contenant la valeur 4 associée à la clef 3
                 # et la valeur 5 associée à la clef "a"
D[3] # renvoie la valeur associée à la clef 3, si elle existe
     # possible erreur KeyError
D[3] = 5 # ajout ou modification de la valeur associée à la clef 3
len(D) # nombre d'éléments dans le dictionnaire
3 in D # expression booléenne valant True si la *clef* 3 est présente dans D
```

 © Pierre Karpman

```
3 not in D # expression booléenne valant True si la *clef* 3 n'est pas
           # présente dans D (surprenamment HORS PROGRAMME)
del(D, 3) # suppression de la clef 3 (si elle est présente)
          # et de la valeur associée. Possible KeyError
          # surprenamment HORS PROGRAMME
```

Les principes généraux du fonctionnement des dictionnaires sont étudiés plus en détail en deuxième année.

§ **Traits du langage**

On présente rapidement deux aspects notables de Python : son système de type et la construction des blocs syntaxiques.

¶ **Typage.** Les types au programme de CPGE sont (cf. ci-dessus) : `int`, `bool`, `float`, `tuple`, `str`, `list`, `dict`. Les trois premiers sont des types de base, et les quatre derniers des types structurés. Le type d'une expression peut être obtenu via le mot-clef `type` (malheureusement **hors programme**) : `type(3)` s'évalue à une valeur de type `type` valant `int`. On peut tester l'égalité entre types : `type(3) == type(4)` s'évalue à `True` tandis que `type(3) == bool` s'évalue à `False`.

En Python (standard), le type d'une expression est déterminé *dynamiquement* à l'exécution. Une conséquence importante est la possibilité d'obtenir des erreurs de type à l'exécution. Par exemple, on peut légalement définir les deux fonctions :

```
def first(L):
    return L[0]

def firstzero():
    return first(0)
```

Mais toute exécution de la seconde entraînera une erreur :

```
TypeError: 'int' object is not subscriptable
```

Dans beaucoup de langages de programmation, la définition de la fonction `firstzero` entraînera elle-même une *erreur de typage* préalablement à son exécution.

Le type d'une variable peut également être modifié au cours d'une exécution. Par exemple, on peut écrire la fonction suivante :

```
def dum(a):
    b = 3
    b = a + b
    if b > 3:
        b = True
    return b # de type (possiblement) int ou bool
```

Encore une fois, dans beaucoup de langages de programmation ceci n'est pas autorisé.

Enfin le typage Python est relativement « faible » : il est possible d'effectuer de nombreuses opérations entre des expressions de types différents, entraînant alors des *conversions* implicites de types. Certaines conventions de conversion peuvent par ailleurs être surprenantes, par exemple :

 © Pierre Karpman

```
In :  2 + True
Out:  3
In :  2 + False
Out:  2
In :  True + False
Out:  1 # ???
In :  D = {1:'wtf'}
In :  D[True]
Out:  'wtf'
```

¶ **Blocs.** Le langage Python est particulièrement original dans la façon dont sont délimités les blocs syntaxiques : ceux-ci le sont par l'*indentation* du code faisant suite à une instruction ou construction définissant un bloc, soit les instructions conditionnelles, les boucles et les définitions de fonction. Les caractères d'indentation (tabulation ou espace) ainsi que leur nombre n'ont pas d'importance, mais doivent être homogènes au sein d'un bloc (et idéalement, au sein de tout un fichier). On illustre l'ensemble de ces notions par le petit exemple suivant :

```
def fun1(a):
    if a > 0: # 4 espaces définissent ici l'indentation des instructions
              # composant le corps de la fonction fun1
        r = 12 % a # 4 espaces définissent les instructions
        r = r * r  # exécutées si la condition du if est vraie
        if r > 0:
            return r # 4 espaces supplémentaires : nouveau bloc
        else: # retour au bloc précédent
            return False # nouveau bloc
    return False
```

Les blocs syntaxiques permettent de définir des *blocs de base* constitués d'instructions qui sont toujours exécutées « ensemble » et par là de construire le *graphe de flot de contrôle*. Ces deux notions sont cependant **hors programme**.

§ Divers

¶ **Entrées/sorties.** Toute expression de type chaîne de caractère peut être affichée sur la *sortie standard* grâce à la fonction `print`. Le système de type de Python (cf. ci-dessus) permet également d'utiliser cette fonction avec des expressions d'autres types, qui seront au préalable converties avant d'être affichées :

```
In :  print('toto')
toto
In :  print(12)
12
```

L'utilisation de l'affichage doit être fait avec une certaine parcimonie. Il est en particulier *très important* de ne pas confondre au sein d'une fonction l'affichage d'une chaîne de caractère avec le fait de renvoyer une valeur (*via* `return`). Même si l'utilisation d'un interpréteur interactif peut laisser penser que ces deux instructions sont interchangeables, elles sont fondamentalement différentes : l'affichage n'est qu'un *effet de bord* qui n'a (généralement ; sauf bugs…) pas d'impact sur le déroulement d'un calcul, tandis qu'une valeur renvoyée est une expression qui peut être utilisée par la suite. On illustre ces différences sur deux exemples :

 © Pierre Karpman

```
In :  print(12)
12    # on affiche la chaîne '12' (convertie depuis l'entier 12)
In :  12
Out:  12 # l'interpréteur *lui-même* affiche la valeur de l'expression 12
```

Soit les deux fonctions :

```
def print12():
    print(12)

def return12():
    return 12
```

On a :

```
In :  print12()
12
In :  return12()
Out:  12
In :  return12() * 2
Out:  24
In :  print12() * 2
TypeError: unsupported operand type(s) for *: 'NoneType' and 'int'
```

En complément de la fonction `print`, les seules fonctionnalités d'entrée/sortie au programme sont la manipulation simple de fichiers textes. On décrit brièvement chacune des fonctions au programme (cette documentation devant être rappelée et éventuellement complétée avant chaque usage : elle n'est pas à connaître par cœur).

— `open` : `f = open('toto.txt', 'r')` « ouvre » le fichier de chemin relatif `'toto.txt'` afin de permettre des lectures (ici) et écritures (cf. ci-dessous). Le premier argument de la fonction est le chemin du fichier à ouvrir, qui est créé s'il n'existe pas encore, et le second est un *mode* d'ouverture. Les modes les plus courants sont :
  — `'r'` : lecture (mode par défaut)
  — `'w'` : écriture depuis zéro : **supprime le contenu du fichier**
  — `'a'` : écriture à la fin du fichier
— `read` : soit `f` un fichier ouvert au préalable en lecture *via* `open`, `f.read()` renvoie une chaîne de caractère contenant le contenu du fichier
— `readlines` : similaire à `read`, mais renvoie une liste des *lignes* du fichier délimitées par le caractère « retour à la ligne » `'\n'`. On peut également directement itérer sur les lignes d'un fichier *via* `for lines in f:` (malheureusement **hors programme**)
— `f.write(s)` : soit `f` un fichier ouvert au préalable en écriture *via* `open`, écrit la chaîne de caractère `s` dans `f` à la position courante de la *tête de lecture* (par défaut, à la fin actuelle du fichier). Attention à éventuellement inclure un caractère « retour à la ligne » dans `s`
— `f.close()` : « ferme » le fichier. Doit être appelé une fois que les traitements (lecture, écriture...) ont été effectués

 © Pierre Karpman

¶ **Interruption d'exécution.** On peut interrompre l'exécution d'un programme (depuis un terminal) ou d'une instruction (dans l'interpréteur interactif) en utilisant la combinaison de touches `Ctrl + C`. Ceci est utile si l'on suspecte que l'exécution ne terminera pas (en un temps raisonnable). Il convient cependant en général d'être patient : tout programme ne se termine pas en moins d'une seconde !

On peut également interrompre l'exécution de n'importe quelle instruction ou fonction *via* l'instruction `assert`. La seule forme au programme de cette instruction est : `assert be` avec `be` une expression booléenne : si `be` s'évalue à `False`, l'exécution de l'instruction ou de la fonction contenant `assert` est interrompue et produit une erreur `AssertionError` ; dans le cas contraire, l'exécution continue normalement. L'utilisation d'assertions est une bonne pratique de programmation : c'est une façon efficace de trouver des erreurs dans un programme et de documenter le comportement attendu.

¶ **Documentation.** On conseille *très vivement* de documenter le code écrit par l'ajout de commentaires, introduits par le caractère `#` (cf. les multiples exemples de ce document). Il n'est pas toujours facile de déterminer le niveau de documentation pertinent, et tous les commentaires ne sont pas utiles (et de là, souhaitables) ; par exemple, un commentaire comme :

```
if a < b : # on teste si a est inférieur à b
```

est complètement inutile : vous (et toute personne vous lisant) est supposée connaître le sens de l'instruction `if` et de l'expression `a < b`. Les commentaires utiles sont souvent ceux qui mettent en évidence des propriétés non triviales par exemples des *invariants* utiles pour prouver la correction du programme :

```
x, y = y, x % y # invariant : gcd(x, y) = gcd(a, b)
```

ainsi que les commentaires expliquant à quelle étape d'un algorithme correspond un certain traitement conséquent (par exemple une suite de boucles imbriquées) ou ceux spécifiant une fonction. En particulier, on conseille vivement de toujours spécifier les types attendus des arguments d'une fonction (ainsi que leurs éventuelles restrictions de valeur) et le traitement effectué. Par exemple :

```
# In:
# L: liste d'int non vide et triée
# x: int
# Out:
# True si x in L, False sinon
def searchdich(L, x)
```

La documentation passe aussi par l'utilisation judicieuse d'assertions (cf. ci-dessus) non nécessairement couplée de commentaires. Par exemple, une fonction contenant l'instruction `assert(len(L) > 0)` documente par son code lui-même le fait qu'à ce point de la fonction, la variable `L` doit être d'un type structuré et doit contenir au moins un élément.

¶ **Mesure de temps de calcul.** Il est souvent (très) instructif de mesure le temps d'exécution d'une instruction, d'une fonction… Ceci peut se faire de façon *relativement simple* en Python : Première solution spécifique à l'interpréteur interactif `ipython` :

```
%timeit es # évalue l'expression e ou exécute
           # l'instruction s, possiblement plusieurs
           # fois et en mesure le temps d'exécution
```

 © Pierre Karpman

Seconde solution, plus générale :

```
# import de la fonction timeit du module timeit
# (à ne faire qu'une fois)
from timeit import timeit
# évalue l'expression ou exécute l'instruction es
# n fois et en mesure le temps d'exécution total
timeit('es', number=n, globals=globals())
```


© Pierre Karpman