

Python : les bases algorithmiques

Introduction

Plan du chapitre

1. Introduction
2. Survol et concepts fondamentaux
3. Principaux types prédéfinis
4. Quelques instructions et sous-programmes
5. Structures de contrôle
6. Exercices de synthèse

Pourquoi Python ?

Pourquoi Python ?

- **open-source**, compatible GPL et utilisations commerciales
- langage **multiplateformes**
- **bibliothèque** très riche et **nombreux modules** :
 - Cryptography, Database, Game Development, GIS (Geographic Information System), GUI, Audio / Music, ID3 Handling, Image Manipulation, Networking, Plotting, RDF Processing, Scientific, Standard Library Enhancements, Threading, Web Development, HTML Forms, HTML Parser, Workflow, XML Processing...
- **importante documentation** :
 - python.org : Tutorial, Language Reference, Library Reference, Setup and Usage
 - wiki.python.org
 - Python Enhancement Proposal (PEP)
 - The Python Package Index (PyPI)
 - [stackoverflow](http://stackoverflow.com)
 - Notions de Python avancées sur Zeste de savoir
 - ...
- **Outils de développement** :
 - IDE (*Integrated Development Environment*) : Idle, Spyder, Pycharm, etc.
 - Documentation : PyDOC, Sphynx, etc.
 - Tests : doctest, unittest, pytest, etc.
 - Analyse statique : pylint, pychecker, PyFlakes, mccabe, etc.
- des **success stories** :
 - Google, YouTube, Dropbox, Instagram,
 - Spotify, Mercurial, OpenStack, Miro, Reddit, Ubuntu...

Quelques précautions à prendre

Deux versions cohabitent Python 2 (2.7) et Python 3 (3.6) : lequel choisir ?

- Compatibilité non préservée entre les deux :
 - exemple : $3 / 2$ donne 1 en Python 2 et 1.5 en Python 3
- Fin de vie Python 2 : 2020, il n'y aura pas de 2.8 ([PEP 373](#), [PEP 404](#))
- Le présent et le futur est Python ≥ 3 (Python 3.0 publiée en 2008 !)

Pas de **typage statique** (mais un typage dynamique fort), il faut donc :

- documenter (et expliciter les choix)
- écrire des tests
- utiliser des analyseurs statiques : `pylint`, etc.
- voir [PEP 0484 \(Type Hints\)](#)

Beaucoup de notions reposent sur des **conventions de nommage**

- noms prédéfinis : `__name__`, `__init__`...
- nom qui commence par un '`_`' signifie susceptible de changer (privé)

Pas de **masquage d'information** (module, classe)

- « nous sommes entre adultes responsables »

Interpréteur Python

Python est langage interprété

L'interprète Python exécute les instructions Python...

On peut l'utiliser de manière **interactive** :

```
$ python
Python 3.5.2+ (default, Sep 22 2016, 12:18:14)
[GCC 6.2.0 20160927] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 3 + 5 ** 2
28
>>> _
```

On peut aussi mettre les instructions Python dans un **fichier** (exemple.py) :

```
# Ceci est un programme python.
print(3 + 5 ** 2)
```

Et l'exécuter en faisant :

```
python3 exemple.py
```

Remarque : L'interpréteur interactif affiche (print) automatiquement les expressions.

Exercices sur l'interpréteur Python

1. Vérifier que l'interpréteur Python est bien installé et contrôler sa version

```
python --version
```

2. Si c'est une version 2 de Python, essayer python3

```
python3 --version
```

3. Si pas de python3,

- l'installer en faisant : `sudo apt-get install python3`
- ou mieux : utiliser `virtualenv` !

4. Utiliser Python comme une calculatrice pour réaliser quelques opérations.

5. Écrire les mêmes opérations dans un fichier (`calculs.py` par exemple) et les exécuter.

Survol et concepts fondamentaux

Objet

Toutes les données manipulées par Python sont des objets : **tout est objet**.

Voici quelques exemples :

```
421      # un entier (int, integer)
3.9      # un nombre réel (float)
4.5e-10   # un autre avec un exposant
3+5j     # un nombre complexe (complex)
"Bonjour" # une chaîne de caractères (str, string)
'x'       # aussi une chaîne de caractères !
[1, 2.5, 'a'] # une liste (list)
```

Un objet possède :

- une **identité** : entier unique et constant durant toute la vie de l'objet.
On l'obtient avec la fonction `id` (l'adresse en mémoire avec CPython)
- un **type** (la classe à laquelle il appartient) que l'on obtient avec la fonction `type`
Le type d'un objet ne peut pas changer.

```
id(421)      # 140067608358512 (par exemple !)
type(421)    # <class 'int'>
type(3.9)    # <class 'float'>
type('x')    # <class 'str'>
```

- des **opérations** généralement définies au niveau de son type

Type

Un **type** définit les caractéristiques communes à un ensemble d'objets.

Parmis ces caractéristiques, on trouve les **opérations** qui permettront de manipuler les objets.

Essayer :

```
dir("str")      # affiche tout ce qui est défini sur `str'  
help("str")     # affiche la documentation de 'str'  
help("str.lower")  # donne la description de la méthode `lower` de `str`.
```

Opération

Parmis les **opérations**, on peut distinguer :

- les **opérateurs « usuels »** : + * / - ** ...
- les **sous-programmes** (procédures¹ ou fonctions) : print, len, id, type, etc.
- les **méthodes** (sous-programmes définis sur les objets) : lower, islower, etc.

Chaque type d'opération a sa **syntaxe d'appel** :

- opérateurs : souvent infixe : 2 * 4
- sous-programme : print(2, 'm') ou len('Bonjour')
- méthode : notation pointée : 'Bonjour'.lower()

```
2 * 4          # 8
2 ** 4         # 16      (puissance)
print(2 ** 4)   # 16      (une procédure)
print(2, 'm')    # 2 m
len('Bonjour')  # 7 : le nombre de caractères de la chaîne  (une fonction)
'Bonjour'.lower() # 'bonjour'      (une méthode)
'Bonjour'.islower() # False        (une autre méthode)
```

1. En Python, tout est fonction !

Nom (ou Variable)

- Une **variable** permet de référencer un objet grâce à un **nom**.
- En Python, on parle plutôt de **nom** que de **variable**.
- Un nom (une variable) est en fait une **référence**, un **pointeur** sur un objet.
- **Intérêt** : nommer les objets (et améliorer la lisibilité du code).

```
x = 5      # x associé à l'entier 5
y = 'a'    # y est associé à la chaîne 'a'
print(x, y) # 5 a    fait référence aux objets associés à x et y
```

- **del** : instruction qui permet de supprimer un ou plusieurs noms.

```
dir()      # 'x' et 'y' dans les noms de l'environnement courant
del x
dir()      # 'x' n'y est plus !
print(x)   # NameError: name 'x' is not defined
print(y)   # 'a'
y = None   # None    objet prédéfini qui signifie sans valeur associée !
y is None  # True    Savoir si y est associé à None
```

- **Remarque** : On a rarement besoin de supprimer explicitement un nom.

Contraintes sur le nom (d'une variable)

Conseil : Choisir un nom, c'est l'occasion de donner du sens à son programme. Il faut donc le choisir avec soin.

Définition : Un nom est un identifiant (lettre ou souligné, suivi de chiffres, lettres ou soulignés).

Convention :

- Un nom de variable est en minuscules.
- Utiliser un souligné _ pour mettre en évidence les morceaux d'un nom : `prix_ttc`
- Voir PEP 8 – Style Guide for Python Code

On ne doit pas utiliser un mot-clé du langage dont voici la liste (`help('keywords')`) :

```
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
```

Attention : Dans un même contexte, on ne peut pas avoir deux variables différentes qui portent le même nom.

Affectation

- **Définition :** L'affectation est l'opération qui permet d'associer un objet à un nom.
- **Syntaxe :** nom = expression avec la sémantique :
 1. évaluer la valeur de l'expression (à droite de =)
 2. associer cette valeur au nom (à gauche de =)

```

prix_ht = 83.25
description = 'Un super produit'
tva = .20
prix_ttc = prix_ht * (1 + tva)      # 99.90 (ou presque : 99.89999999999999)
prix_ht = 100.0
prix_ttc                         # toujours presque 99.90

a = b = c = 0          # a, b et c associés l'objet 0

```

- Affectation multiple : le ième nom est associé au ième objet :

```

nom, age = 'Paul', 18 # équivalent de nom = 'Paul'; age = 18
a, b, c = 1, 2 ** 3, -1      # a == 1 and b == 8 and c == -1
a, b, c = c, a, b          # a == -1 and b == 1 and c == 8

```

- Une variable n'a pas de type propre : elle a le type de l'objet qu'elle référence.

```

type(age)    # <class 'int'>
type(nom)    # <class 'str'>

```

Typage dynamique

Le « type d'une variable » peut changer... Il suffit de l'affecter avec un objet d'un autre type !

```
x = 5      # type(x) : int
x = 2.3    # type(x) : float
x = 'non'  # type(x) : str
```

Python s'appuie exclusivement sur du **typage dynamique** contrairement au langage C (ou Pascal, Ada, Java, C++, Caml, etc.) qui s'appuie sur du **typage statique**.

```
int x = 5; // int est le type de déclaration de la variable x
x = 2.3;  // refusé par le compilateur, avant l'exécution
x = "non"; // idem.
```

Attention : Seules les erreurs de syntaxes sont détectées lors du chargement d'un programme par l'interpréteur Python. Les autres erreurs sont signalées à l'exécution. **Il faut donc tester !**

Pour chaque variable, choisir un nom significatif, préciser son type (commentaire) et s'y tenir !

Pour savoir si un objet est d'un certain type, on peut utiliser `isinstance` :

x = 421	isinstance(x, int) # True	isinstance(x, float) # False
	isinstance(x, str) # False	isinstance(x, bool) # False

Bien comprendre Nom et Objet : le partage

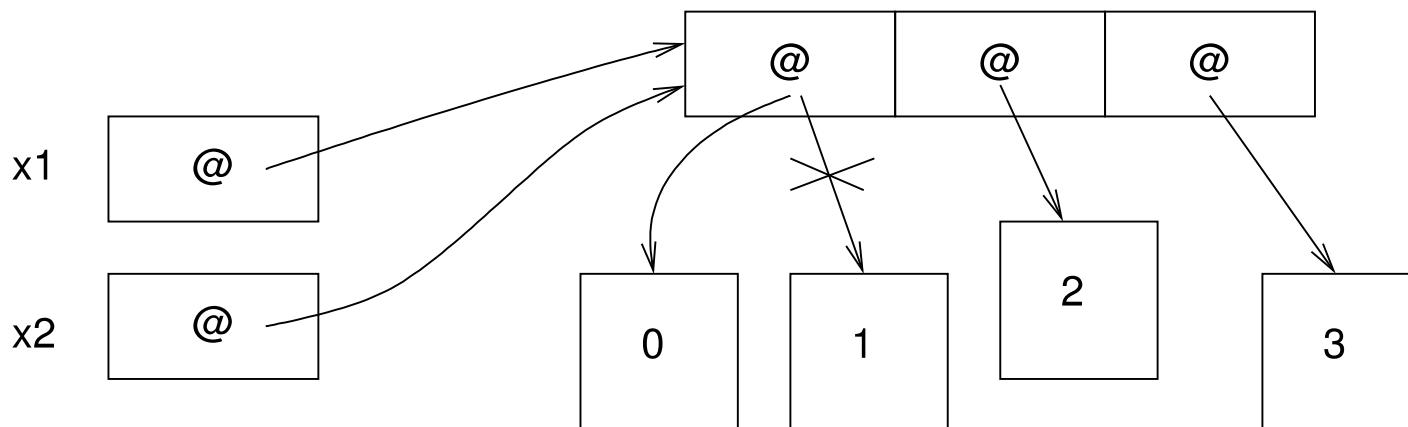
- Un objet ne peut changer ni d'identité, ni de type.
- Plusieurs noms peuvent référencer le même objet (ce sont des **alias**).
- Tout changement fait sur l'objet depuis l'un des noms est visible depuis autres noms !

```

1  x1 = [1, 2, 3]      # une liste contenant 3 éléments 1, 2 et 3
2  x2 = x1            # un deuxième nom sur la même liste
3  x1[0] = 0          # changement du premier élément de la liste
4  print(x2)          # [0, 2, 3]
5  id(x1) == id(x2)   # True : x1 et x2 donnent accès au même objet

```

- Un nom est équivalent à un pointeur, une référence sur un objet



Égalité physique (`is`) et égalité logique (`==`)

Définitions :

- **Égalité logique** : deux objets (éventuellement différents) ont mêmes valeurs
- **Égalité physique** : deux noms référencent le même objet (même identifiant)

```
x1 = [1, 2, 3]      # une liste contenant 3 éléments 1, 2 et 3
x2 = x1              # un deuxième nom sur la même liste
x3 = [1, 2, 3]      # une autre liste contenant 1, 2 et 3
x1 is x2            # True
x1 is x3            # False
x1 == x2            # True
x1 == x3            # True
x3[0] = 0
x1 == x3            # False
x1 != x3            # True, négation de ==
x1 is not x3        # True, négation de is
```

- L'opérateur `is` teste l'**égalité physique** : même objet
- L'opérateur `==` teste l'**égalité logique** : mêmes valeurs
- `n1 is not n2` est équivalent à `not (n1 is n2)`
- `n1 != n2` est équivalent à `not (n1 == n2)`
- **Normalement** : égalité physique implique égalité logique (exception : NaN, i.e. `math.nan`)

Remarque : `n1 is n2` est équivalent à `id(n1) == id(n2)`. Préférer `is` !

Objet mutable et objet immuable

Définition :

- Objet **immutable** : objet dont l'état ne peut pas changer après sa création.
- Objet **mutable** : objet dont l'état peut changer au cours de sa vie.
- Synonymes : altérable/inaltérable, modifiable/non modifiable (anglais : *mutable/imutable*)

```
s1 = 'bon'          # Les chaînes sont immuables :
s1[0] = 'b'        # TypeError: 'str' object does not support item assignment
del s1[0]          # TypeError: 'str' object doesn't support item deletion
s2 = s1.lower()    # Création d'un nouvel objet
s2 is s1           # False      on a bien deux objets différents
s2 == s1           # True       mais logiquement égaux

l1 = [ 1, 2, 3 ]   # Les listes sont muables, la preuve :
l1[0] = -1         # [-1, 2, 3]
del l1[0]          # [2, 3]
l1.append(4)        # [2, 3, 4]
```

Objet immuable vs objet mutable :

- Un objet immuable peut être partagé sans risque car personne ne peut le modifier
- ... y compris depuis des fils d'exécution différents (threads)
- Mais chaque « modification » nécessite la création d'un nouvel objet !

Principaux types prédéfinis

Types numériques

Quelques types numériques

entier (int)

- entiers relatifs (positifs ou négatifs)
- exemples : -153, 0, 2048
- arbitrairement grands

```
2 ** 200 # 1606938044258990275541962092341162602522202993782792835301376
```

réel (float)

- nombres à virgule flottante
- exemples : 4.5 5e128 -4e-2 1.12E+30
- exposant de -308 à +308, précision 12 décimales, Voir IEEE 754

Opérateurs

opérateurs arithmétiques

opération	résultat	exemple avec $x = 10; y = 3$
$x + y$	somme	13
$x * y$	produit	30
$x - y$	soustraction	7
x / y	division réelle	3.333333333333335
$x // y$	division entière	3
$x \% y$	reste de la division entière	1
$-x$	opposé	-10
$+x$	neutre	10
$x ** y$	puissance	1000
$abs(x)$	valeur absolue	$abs(-10)$ donne 10
$int(x)$	conversion vers un entier	$int(3.5)$ donne 3, comme $int('3')$
$float(x)$	conversion vers un réel	$float(10)$ donne 10.0, comme $float('10')$

Forme contractée de l'affectation

$x \# = y$ est équivalent à $x = x \# y$

Exemple : $x += 2$ est équivalent à $x = x + 2$

Opérateurs relationnels

```
<  <=  >  >=  ==  !=  is  is not
```

Opérateurs sur les bits

```
|  ^  &  <<  >>  ~
```

Opérateurs logiques

and or not

évaluation en court-circuit : n != 0 and s / n >= 10

Formules de De Morgan

`not (a and b) <==> (not a) or (not b)`

`not (a or b) <==> (not a) and (not b)`

`not (not a) <==> a`

Tout objet peut être considéré comme booléen.

Il sera faux, s'il est nul, de longueur nulle, etc et vrai sinon.

Voir `__bool__()`.

Priorité des opérateurs (priorité faible en haut)

P	Operator	Description
1	"lambda"	Lambda expression
2	"if" – "else"	Conditional expression
3	"or"	Boolean OR
4	"and"	Boolean AND
5	"not" "x"	Boolean NOT
6	"in", "not in", "is", "is not", "<", "<=", ">", ">=", "!=" , "=="	Comparisons, including membership tests and identity tests
7	" "	Bitwise OR
8	"^"	Bitwise XOR
9	"&"	Bitwise AND
10	"<<", ">>"	Shifts
11	"+", "-"	Addition and subtraction
12	"*", "@", "/", "//", "%" "	Multiplication, matrix multiplication division, remainder [5]
13	"+x", "-x", "~x"	Positive, negative, bitwise NOT
14	"**"	Exponentiation [6]
15	"await" "x"	Await expression
16	"x[index]", "x[index:index]", "x(arguments...)", "x.attribute"	Subscription, slicing, call, attribute reference
17	"(expressions...)", "[expressions...]", "{key: value...}", "{expressions...}"	Binding or tuple display, list display, dictionary display, set display

Exercices

1. Lire les indications données par `help("OPERATORS")` ou `help("//")`
2. Parenthésier les expressions suivantes

1 `x != y * 3 + 5`
2 `n != 0 and s / n >= 10`
3 `y >= b and z ** 2 != 5 and not x in e`

1. Comment s'évaluent les expressions suivantes :

1 `x = 12`
2 `y = 2 ** 2 ** 3`
3 `not x and y > 0`

2. Que signifient les expressions suivantes ?

1 `x < 10 < y`
2 `x < 10 >= y`
3 `x < 10 == y`

Conseil : Éviter d'écrire des expressions trop compliquées, utiliser des variables !

le type str (chaîne de caractères) : sequence immuable et ordonnée de caractères

Chaînes littérales

- Utiliser apostrophe ('') ou guillemet ("") pour délimiter les chaînes

```
s1 = "Bonjour"          # avec des guillemets
s2 = 'Bonjour'          # avec apostrophe
s3 = 'Bon' "jour"       # concaténation implicite
s4 = 'Bon' + "jour"     # concaténation explicite
s5 = 'Bon' \
    "jour"      # \ : caractère de continuation
s6 = """Une chaîne
        sur plusieurs
        lignes"""
s7 = '''Avec une ' (apostrophe)
        dedans'''
s8 = "des caractères spéciaux : \t, \n, \\", \'...
s9 = 'valeur : ' + str(12) # conversion explicite en str()
```

- Variante avec **triple délimiteur** : permet de continuer la chaîne sur plusieurs lignes.
- Les chaînes avec triple délimiteurs sont utilisées pour la **documentation**.
- Il n'y a **pas de type caractère** en Python (idem chaîne de longueur 1)

Opérateur sur str (comme séquence immuable)

Séquence : plusieurs éléments répérés par une position, un indice.

Opération	Résultat	Exemples
<code>x in s</code>	True si x est une sous-chaîne de s	<code>'bon' in 'bonjour'</code>
<code>x not in s</code>	True si x n'est pas une sous-chaîne de s	<code>'x' not in 'bon'</code>
<code>s + t</code>	concaténation de s avec t	
<code>s * n ou n * s</code>	équivalent à ajouter s à elle-même n fois	<code>'x' * 3 donne 'xxx'</code>
<code>s[i]</code>	ième caractère de s, i == 0 pour le premier	<code>'bon'[-1] donne 'n'</code>
<code>len(s)</code>	la longueur de s (nombre de caractères)	<code>len('bon') donne 3</code>
<code>min(s)</code>	plus petit caractère de s	<code>min('onjour') donne 'j'</code>
<code>max(s)</code>	plus grand caractère de s	<code>max('onjour') donne 'u'</code>
<code>s.index(x[, d[, f]])</code>	indice de la première occurrence de x dans s (à partir de l'indice d et avant l'indice f)	<code>'bonjour'.index('o') donne 1</code> <code>'bonjour'.index('o', 2) donne 4</code>
<code>s.count(x)</code>	nombre total d'occurrence de x dans s	<code>'bonjour'.count('o') donne 2</code>

- i est un indice valide sur sssi $-\text{len}(s) \leq i < \text{len}(s)$, sinon IndexError !
- `'bonjour'.index('o', 2, 4)` lève l'exception ValueError car non trouvé
- **Remarque** : Toutes ces opérations sont présentes sur toute séquence.
- Les chaînes de caractères sont des **séquences immuables** de caractères.

Méthodes spécifiques de str

```

s = ' ET '.join( [ 'un' , 'deux', 'trois' ] )      # concaténer avec ce séparateur
s
          # 'un ET deux ET trois'

'chat' < 'chien'      # True      (Ordre lexicographique)
'chat' < 'chats'     # True

ord('0')              # 48... mais quelle importance ?
                      # ord permet d'obtenir le code d'un caractère
chiffre = 5
c = chr(ord('0') + chiffre)    # le caractère correspondant à un code
c
          # '5'
'5'

r = ' xx yy  zz '.strip()    # supprimer les blancs du début et de la fin
r
          # 'xx yy  zz'

r = ' xx yy  zz '.split()    # découper une chaîne en liste de chaînes
r
          # ['xx', 'yy', 'zz']

```

Mais aussi lower, islower, upper, isupper, replace, etc. Voir help('str').

Exercices

1. Comment obtenir le dernier caractère d'une chaîne ?

Exemple : 'bonjour' -> 'r'

2. Remplacer tous les 'e' d'une chaîne par '*'.

Exemple : 'une chaîne' -> 'un* châin*'

Idem avec 'ne' deviennent '...'.

Exemple : 'une chaîne' -> 'u... châi...'

3. Étant donné une chaîne et un caractère, trouver la position de la deuxième occurrence de ce caractère dans la chaîne.

Exemple : 'bonjour vous' et 'o' -> 4

4. Indiquer combien il y a de mots dans une chaîne de caractères.

Exemple : 'bonjour vous' -> 2

Exemple : 'il fait très beau' -> 4

5. Indiquer le nombre d'occurrences d'une lettre dans une chaîne.

Exemple : "C'est l'été, n'est-ce pas ?" contient 1 'a', 3 'e', 0 'v', 3 "", 2 'st', etc.

Le type list (liste) : séquence modifiable

list : séquence modifiable

Création d'une liste

```
l1 = [ 1, 2, 3]      # liste [1, 2, 3]
l2 = []              # une liste vide
```

Opérations sur une liste

```
l1[0]                  # l1 : premier élément de la liste
len(l1)                # 3
len(l2)                # 0
l1 += [4, 5]            # l1 == [1, 2, 3, 4, 5]  # concaténer
del l1[3]               # l1 == [1, 2, 3, 5]
l1[1] = 'x'             # l1 == [1, 'x', 3, 5]  # hétérogène !
l1.append('x')          # l1 == [1, 'x', 3, 5, 'x']
l1.count('x')           # 2
l1.insert(1, True)       # l1 == [1, True, 'x', 3, 5, 'x']
l1.remove('x')           # l1 == [1, True, 3, 5, 'x']
p, *m, d = l1            # p == 1 and m == [True, 3, 5] and d == 'x'
```

Séquence modifiable

Une liste est une séquence modifiable. Elle a donc toutes les opérations d'une séquence immuable + des opérations de modification.

Exercices

Indiquer, après l'exécution de chaque ligne, la valeur de la liste l.

```
l = []
l.append(2)
l.insert(0, 4)
l.insert(2, 1)
l[1] = 'deux'
l[2] /= l[2]
l.count(1)
l[0], l[1] = l[1], l[0]

p, _, t = l      # p ? _ ? d ?
premier, *suite = l # premier ? suite ?

b = [False, True]
l += b
l2 = [2, 3, 5]
i, l2[i], *_ = l2    # l2 ?
l.append(l2)
l2.append(l)          # l2 ?

l = list('Fin.')
```

Le type tuple (tuple) : séquence immuable

Tuple : une séquence immuable

- Un tuple (n-uplet) permet d'assembler plusieurs objets dans une séquence immuable.

```
t = (1, 'premier', 10.5)      # t est un tuple composé de 3 objets
a, b, c = t # déstructurer le tuple : a == 1, b == 'premier', c == 10.5
_, x, _ = t # x == 'premier'
t[0]          # 1
t[1]          # 'premier'
t[2]          # 10.5
t[-1]         # 10.5
t = 1, 'premier', 10.5       # les parenthèses peuvent être omises (si pas ambigu)
t = (1, )    # tuple avec un seul élément (virgule obligatoire)
```

- Représenter une date avec le numéro du jour, du mois et de l'année :

```
date_debut = (21, 11, 2016) # ici, on choisit (jour, mois, année). À documenter !
```

- On peut construire un tuple à partir d'une séquence

```
tuple('abc')    # ('a', 'b', 'c')
tuple([1, 'X']) # (1, 'X')
```

- Tuple ou liste ? Les deux sont des séquences (donc opérations communes) mais

- liste modifiable et tuple immuable
- tuple : les éléments ont un sens en fonction de leur position
- liste : les éléments sont interchangeables

range : séquence immuable d'entiers

- range permet de définir des séquences immuables d'entiers.
- Il est généralement utilisé pour les répétitions (for).
- Appels possibles :

```
range(stop) -> range object  
range(start, stop[, step]) -> range object
```

- Quelques exemples :

```
r1 = range(4)  
r1                      # range(0, 4)  
list(r1)                 # [0, 1, 2, 3]  
tuple(r1)                # (0, 1, 2, 3)  
r1[-1]                   # 3  
  
list(range(4, 8))         # [4, 5, 6, 7]  
list(range(2, 10, 3))     # [2, 5, 8]  
list(range(5, 2, -1))     # [5, 4, 3]  
list(range(2, 3, -1))     # []
```

Les slices (tranches)

Motivation : Référencer une partie des objets contenus dans une séquence.

Forme générale : sequence [debut:fin:pas]

- les éléments de sequence de l'indice debut inclu, à l'indice fin exclu avec un pas
- les indices peuvent être positifs (0 pour le premier élément) ou négatifs (-1 pour le dernier)
- possibilité d'utiliser les opérateurs classiques : **del**, **=**, etc.

Exemples :

```
s = list(range(10))      # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
s[2:4]                  # [2, 3]
s[2:]                   # [2, 3, 4, 5, 6, 7, 8, 9]
s[:-4]                  # [0, 1, 2, 3, 4, 5]
s[2::4]                 # [2, 6]
s[::-1]                  # [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
s[8:2:-2]               # [8, 6, 4]      (parcours de la fin vers le début)

del s[2::2]              # [0, 1, 3, 5, 7, 9]      (supprimer les éléments de la slice)
s[-4:] = s[:4]            # [0, 1, 0, 1, 3, 5]
s[:-1] = s[1:]            # [1, 0, 1, 3, 5, 5]
s[1:] = s[:-1]            # [1, 1, 0, 1, 3, 5]
s[1:3] = [9, 8, 7, 6]     # [1, 9, 8, 7, 6, 1, 3, 5]
s[::2] = list(range(9))   # ValueError: attempt to assign sequence of size 9 to extended slice of s

range(0, 4)[::-1]         # range(3, -1, -1)
```

Pour en savoir plus...

Exercices

1. Comment représenter une carte d'un jeu de 52 cartes ?
2. Comment représenter un jeu de 52 cartes ?
3. Étant donnée une liste l , comment désigner la liste de tous les éléments sauf le premier et le dernier ?
Exemple : $l = [2, 3, 4, 5]$ donne $[3, 4]$
4. Étant donnée une liste l , comment obtenir deux listes, la première qui contient les éléments d'indice pair et la seconde les éléments d'indice impair ?
Exemple : $l = [-5, 2, 1, 18, 0]$ donne $[-5, 1, 0]$ et $[2, 18]$
5. Comment représenter une tâche caractérisée par un nom, une priorité, une difficulté et une date d'échéance ?

Quelques instructions et sous-programmes

Instruction pass

Instruction pass

Définition

L'instruction **pass** est une instruction qui ne fait rien.

Intérêt

Elle est utile quand une instruction est attendue syntaxiquement... mais qu'aucun code n'est nécessaire (pour l'instant...).

Instruction assert

Instruction assert

Objectif

L'instruction **assert** est un moyen simple de vérifier que les hypothèses faites lors de la conception et l'implantation du programme sont effectivement respectées pendant son exécution.

Exemples

```
n = 0
assert n > 0      # Lève une exception AssertionError si la condition est fausse
                   # affiche : ... AssertionError
assert n > 0, 'n == ' + str(n)  # avec un objet qui sera associé à l'AssertionError
                               # affiche : ... AssertionError: n == 0

assert condition, message    # forme générale
```

Désactiver les instructions assert

- par défaut, les assert sont vérifiées en Python
- pour les désactiver, lancer l'interpréteur Python avec l'argument `-O`

Conseil

- Ne jamais écrire un programme dont l'exécution exploite le résultat de assert

Instructions d'entrée/sortie (print et input)

Instruction de sortie : print

- **print()** permet d'écrire un ou plusieurs objets :

- sur la sortie standard
- en écrivant un espace entre chaque objet (*sep*)
- en écrivant un retour à la ligne (*end*)

```
a, b = 18, 'ok'  
print('a =', a, 'et b =', b)      # a = 18 et b = ok  
print('a =', a, 'et b =', b, sep='_', end='~!\n')  
      # a =_18_et b =_ok~!
```

- la méthode **str.format()** permet de simplifier les écritures

```
print('a = {} et b = {}'.format(a, b))  # a = 18 et b = ok  
      # les {} correspondent aux paramètres de format  
print('{0}, {1}, {0}'.format(a, b)) # 18, ok, 18  
      # le numéro entre {} est le numéro du paramètre à écrire  
print('{:7.2f}'.format(1.2345))      #    1.23 (avec des espaces devant)  
      # 7 caractères dont 2 pour la partie décimale
```

```
# Et avec python >= 3.6, un f devant une chaîne la traite comme format :  
nombre, largeur, precision = 1.2345, 10, 3  
print(f"nombre = {nombre:{largeur}.{precision}f}") # nombre = 1.23
```

Instruction de saisie : `input`

- `input()` permet de demander à l'environnement du programme une information sous la forme d'une chaîne de caractères.

```
>>> reponse = input('Votre choix ? ')
Votre choix ? quitter
>>> reponse
'quitter'
```

- Si on attend un entier ou réel, il faudra convertir la chaîne obtenue

```
>>> reponse = input('Un entier : ')
Un entier : 15
>>> n = int(reponse)      # conversion d'une chaîne en entier
>>> n
15
>>> n = float(input())   # pas de prompt
4.5      # Chaîne saisie
>>> n
4.5
```

Attention : Risque de `ValueError` si la chaîne ne correspond pas à un entier (resp. un réel).

Structures de contrôle

Structures de contrôle

Objectif

Les **structures de contrôle** décrivent l'ordre dans lequel les instructions seront exécutées.

- enchaînement séquentielle (bloc ou séquence),
- traitements conditionnels (**if ... elif ... else**)
- traitements répétitifs (**while** et **for**)
- mécanisme d'exception
- appel d'un sous-programme (vu dans un autre chapitre)

Bloc

Bloc

Bloc

Un **bloc** est une suite d'instructions qui sont exécutées dans l'ordre de leur apparition.

Synonymes : **séquence** ou **instruction composée**

Règle

- Toutes les instructions d'un même bloc doivent avoir exactement la même indentation.

Conseil

Ne pas mélanger espace et tabulation dans les indentations. Python recommande de n'utiliser que les espaces et une indendation de 4 espaces.

conditionnelle if

conditionnelle if (Si)

Conditionnelle if . . . else . . .

```
if condition:  
    blocSi  
else:  
    blocElse
```

- Les deux-point (":") en fin de ligne introduit un bloc. L'indentation doit être augmentée.
- La condition est évaluée.
- Si elle est vraie alors *blocSi* est exécuté
- Si elle est fausse alors *blocElse* est exécuté

Propriétés

- Les deux blocs sont exclusifs
- La partie **else** est optionnelle

Exemple

```
if n1 == n2:  
    résultat = 'égaux'  
else:    # n1 != n2  
    résultat = 'différents'
```

Exemple sans else

```
if n > max:  
    max = n
```

Exercice : Année bissextile

Étant donné un entier, afficher “bissextile” s'il correspond à une année bissextile et “non bissextile” sinon.

Partie **elif** (SinonSi)

SinonSi : elif

On peut ajouter après le **if** des **elif** (SinonSi) de la forme suivante :

```
if condition1:  
    assert condition1  
    bloc1  
elif condition2:  
    assert not condition1 and condition2  
    bloc2  
...  
elif conditionN:  
    assert not condition1 and ... and not conditionN-1 and conditionN  
    blocN  
else:  
    assert not condition1 and ... and not conditionN  
    blocE  
suite
```

- Les conditions sont évaluées dans l'ordre
- Pour la première condition vraie, le bloc associé est exécuté, puis l'exécution continue à 'suite'
- Si aucune condition n'est vraie, le *blocE* du **else** est exécuté puis l'exécution continue à 'suite'

Exemple

```
if n > 0:  
    resultat = 'positif'  
elif n < 0:      # not (n > 0) and (n < 0)  
    resultat = 'negatif'  
else:            # not (n > 0) and not (n < 0)  
    resultat = 'nul'
```

Exercice : Classer un caractère

Étant donné un caractère c (chaîne d'un seul caractère), indiquer s'il s'agit d'une voyelle, d'une consonne, d'un chiffre ou d'un autre caractère.

Exercice : Nombre de jours d'un mois

Étant donné un numéro de mois compris entre 1 et 12, détermine son nombre de jours.

Formes équivalentes au `if`

Formulations à éviter !

```
if condition:
    resultat = True
else:
    resultat = False
```

| `if condition:`
| `resultat = False`
| `else:`
| `resultat = True`

Les formulations précédentes, à éviter, peuvent être réécrites avec une simple expression booléenne :

<code>resultat = condition</code>		<code>resultat = not condition</code>
-----------------------------------	--	---------------------------------------

Exemple : Un individu est *majeur* ssi son *age* est supérieur ou égal à 18.

Si Arithmétique

```
if condition:
    resultat = valeurVrai
else:
    resultat = valeurFaux
```

peut se réécrire en :

```
resultat = valeurVrai if condition else valeurFaux
```

Intérêt : Il s'agit d'une expression et non d'une instruction

Critique : Lisible ?

Répétition TantQue (while)

Répétition while (TantQue)

Forme

```
while condition:  
    bloc
```

Propriétés

- **Sémantique** : Tant que *condition* est vraie, *bloc* est exécuté.
- **Remarque** : On peut ne pas exécuter *bloc* : *condition* est fausse au départ
- **Règle** : Il faut que *bloc* modifie *condition* (sinon boucle infinie ou code mort)

Exemple : somme des n premiers entiers

```
n = 10  
i = 1      # pour parcourrir les entiers de 1 à n  
somme = 0  # la somme des entiers  
while i <= n:  # i est à prendre en compte  
    somme = somme + i      # ou somme += i  
    i = i + 1              # ou i += 1  
print('la somme des entiers de 1 à {} est : {}'.format(n, somme))
```

Exercice : Exécuter à la main ce programme avec $n = 3$

Question : Peut-on faire ce calcul plus efficacement ?

Terminaison et correction de boucles

Problème

Contrairement au code déjà écrit, on peut revenir en arrière dans le programme.

Comment être sûr qu'il va s'arrêter ?

```
i = 1  
while i < 10:  
    print(i)
```

Est-ce que ce programme s'arrête ? Pourquoi ? Le corriger. Comment montrer sa terminaison ?

Terminaison

Variant : Quantité entière positive qui décroît strictement à chaque passage dans la boucle.

Variant de "somme des n premiers entiers" : $n - i + 1$ (nombre d'entiers restant à sommer)

Il faut identifier le variant et démontrer ses propriétés : entier, positif, décroissance stricte.

Correction d'une répétition : Variant et invariant

Invariant : Propriété toujours vraie, avant et après chaque passage dans la boucle

Invariant de "somme des n premiers entiers" : somme == somme des entiers de 0 à $i - 1$

Exercice : Montrer que c'est un invariant ?

Conseil : A défaut de formaliser l'invariant, l'indiquer dans un commentaire.

Exercice : A-t-on toutes les informations pour montrer la correction ?

continue, break et else

Forme générale du while

```
while condition:  
    bloc # pouvant inclure les instructions 'continue' et 'break'  
else:  
    blocAutre
```

- **continue** : passe directement à l'itération suivante de la boucle englobante
- **break** : sort de la boucle englobante sans exécuter le bloc du **else**
- **else** : le bloc associé est exécuté à la fin de la boucle ssi pas de **break** (ni **return**)

Exemple

```
n, max = 0, 5  
while n < max:  
    n += 1  
    if n % 2 == 0:  
        continue  
    elif n > 10:  
        print('abandon')  
        break  
    print(n)  
else:  
    print('fin')
```

- Donner le résultat de l'exécution avec `n, max = 0, 5`
- Donner le résultat de l'exécution avec `n, max = 0, 50`

Exercices

Exercice : Nombre de chiffres dans une chaîne

Écrire un programme qui, étant donnée une chaîne de caractères, affiche le nombre de chiffres (caractère de '0' à '9') qu'elle contient. Exemple : 'Le prix de 10 lots de 5 vaut 45.' donnera 5.

Exercice : Afficher les lettres et chiffres d'une chaîne

Écrire un programme qui, étant donnée une chaîne de caractères, n'affiche à l'écran que les caractères qui sont des lettres, minuscules ou majuscules, des espaces ou des chiffres.

Exemple : 'Un texte, avec 1 vigule et autres!' s'affichera 'Un texte avec vigule et autres'

L'améliorer pour n'afficher qu'un seul espace au lieu de plusieurs consécutifs.

Exercice : Indice d'un caractère dans une chaîne

Écrire un programme qui étant donnés un chaîne de caractères et un caractère, indique la position de la première occurrence de ce caractères dans la chaîne ou "pas trouvé" sinon.

On n'utilisera pas les méthodes de str, seulement l'accès au ième élément (notation s[i]).

Exercice : nombre d'occurrence d'un élément dans une liste

Écrire un programme qui étant donnés une liste et un élément, calcule et affiche le nombre d'occurrence de cet élément dans cette liste.

On n'utilisera pas la méthode 'count' de 'list'.

Répétition PourChaque (for)

Répétition for (PourChaque)

Forme

```
for nom in expression:  
    bloc
```

Exemple

```
liste = [ 1, 2, 3, 4, 5 ]  
for nombre in liste:  
    print( nombre ** 2, end=' ' )  
donne '1 4 9 16 25 '
```

Propriétés

Sémantique :

- *expression* doit être une *séquence*^a, par exemple list, tuple, str...
- *nom* prend successivement chaque valeur de l'*itérable*
- *bloc* est exécuté pour chaque affectation de *nom*

Principe :

- On sait combien de fois on exécute *bloc* (autant que d'éléments dans *expression*)
- La terminaison est donc garantie... si la séquence est finie.

a. En fait, ce doit être un *itérable*.

Autres éléments utiles : **range**, **enumerate**

range : séquence d'entiers (voir `help('range')`)

```
for nombre in range(0, 10): # essayer avec range(0, 10, 3)
    print(nombre, end=' ')
affiche '0 1 2 3 4 5 6 7 8 9 '
```

Plusieurs noms

```
liste = [ (1, 'un'), (2, 'deux') ]
for numero, texte in liste:
    print(f'{numero} -> {texte}', end=' ; ')
affiche : '1 -> un ; 2 -> deux ; '
```

enumerate : ajouter un numéro d'ordre (voir `help('enumerate')`)

```
tuple = ( 'un', 'deux', 'trois' )
for numero, texte in enumerate(tuple, 1):
    print('{} -> {}'.format(numero, texte), end=' ; ')
affiche : '1 -> un ; 2 -> deux ; 3 -> trois' ;
```

for, continue, break et else

for admet une clause else (comme while)

```
for noms in expression:
```

```
    bloc
```

```
else:
```

```
    blocElse
```

Même sémantique que le while

Exercices

Essayer d'écrire avec un for les exercices faits avec while. Plus simple ?

while ou for

Donner des éléments pour aider à choisir entre un while et un for.