Python : les bases algorithmiques

DAFS : Algorithmique et programmation

Xavier Crégut <prenom.nom@enseeiht.fr>

Introduction

Plan du chapitre

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
  - Notions de Python avancées sur Zeste de savoir
  - ...
- **Outils de développement :**
  - IDE (*Integrated Development Environment*) : Idle, 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) : le quel chosir ?

- 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 !)

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 »

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

- documenter
- écrire des tests
- utiliser des analyseurs statiques : pylint, etc.
- voir PEP 0484 (Type Hints)

Langage **très dynamique** !

- peut être déroutant quand on vient d'un langage typé statiquement (C, Ada, Java, C++, Pascal, Caml, etc.)

Interpréteur Python

Python est langage interprété

L'interprête 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'interprèteur interactif affiche (`print`) automatiquement les expressions.

Exercices sur l'interpréteur Python

1. Vérifier que l'interprêteur 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 carctères !
[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 a 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('Bonjour') # <class 'str'>
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** (procédures 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)
```

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 ** 2  # Les noms ne sont pas significatifs ici !
y = 3 + x
print(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)    # 28
y = None    # None : objet prédéfini qui signifie sans valeur associée !
y is None   # True
```

- **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, 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.

```
prix_ht = 83.25
description = 'Un super produit'
a = b = c = 0        # a, b et c référencent l'objet 0
```

- Affectation multiple : le ième nom référence le ième objet :

```
nom, age = 'Paul', 18   # équivalent de nom = 'Paul'; age = 18
a, b = 1, 2     # a == 1 and b == 2
a, b = b, a     # a == 2 and b == 1
```

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

```
type(89.90) == type(prix_ht)    # True
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 dyamique** 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;    // interdit par le compilateur, avant l'exécution
x = "non";  // idem.
```

Risque d'erreur : Que penser de l'expression `langue = 'FR'; print(langue.lower == 'fr')` ?

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

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

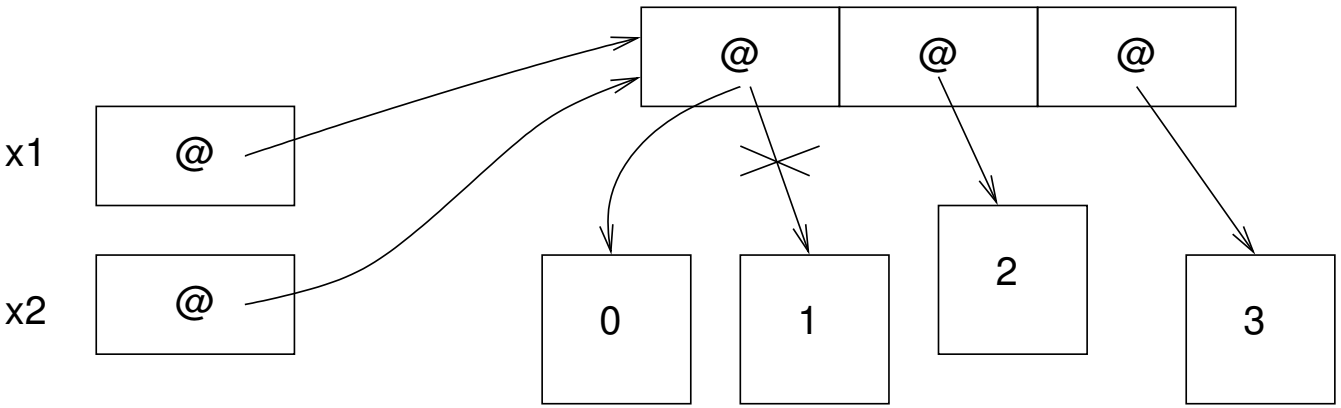
```
x = 421
isinstance(x, int)   # True
isinstance(x, str)   # False
isinstance(x, float)    # False
isinstance(x, bool) # False
```

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 (==)

```
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)

Objet muable et objet immuable

Définition :

- Objet **immuable** : objet dont l'état ne peut pas changer après sa création.
- Objet **muable** : 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 = 'Bonjour'.lower()  # Création d'un nouvel objet
s2 is s1          # False

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 muable :

- Un objet immuable peut être partagé sans risque car personne ne peut le modifier
- ... y compris depuis des flots 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	3.3333333333333335
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. Parenthéser les expressions suivantes

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

1. Comment s'évaluent les expressions suivantes :

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

2. Que signifient les expressions suivantes ?

```
x < 10 < y
x < 10 >= y
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 litérales

- 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)  # convertion explicite en str()
```

- Variante avec **triple délimiteur** qui 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 sequence immuable)

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

- `'bonjour'.index('o', 2, 4)` lève l'exception ValueError
- **Remarque :** Toutes ces opérations sont présentes sur toute séquence.
- Les chaînes de caractères sont des **séquences immuables** de caractères.

Méthodes spécifiques de str

```
>>> ' et '.join( [ 'un' , 'deux', 'trois' ] )   # concaténer avec ce séparateur
'un et deux et trois'

>>> 'chat' < 'chien'        # Ordre lexicographique.
True

>>> 'chat' < 'chats'
True

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

>>> '   xx yy   zz  '.strip()   # supprimer les blancs au début et à la fin
'xx yy  zz'

>>> '   xx yy   zz  '.split()
['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* chaîn*'`
   Idem avec `'ne'` deviennent `'...'`.
   Exemple : `'une chaîne'` -> `'u... chaî...'`
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]                # 1 : premier élément de la liste
len(l1)              # 3
len(l2)              # 0
l1 += [4, 5]         # concaténer : [1, 2, 3, 4, 5]
del l1[3]            # [1, 2, 3, 5]
l1[1] = 'x'          # [1, 'x', 3, 5]    # hétérogène !
l1.append('x')       # [1, 'x', 3, 5, 'x']
l1.count('x')        # 2
l1.insert(1, True)   # [1, True, 'x', 3, 5, 'x']
l1.remove('x')       # [1, True, 3, 5, 'x']
```

Séquence modifiable

Une liste est une séquence modifiable. Elle a dont 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 # destructurer 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 élements 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='.FIN.\n')
    # a =...18...et b =...ok.FIN.
```

- 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 une 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 méme bloc doivent avoir exactement la même indentation.

conditionnelle if

conditionnelle **if** (Si)

Conditionnelle if . . . else . . .

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

- Le 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:
    resultat = 'égaux'
else:   # n1 != n2
    resultat = '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:
    bloc1
elif condition2:     # not condition1 and condition2
    bloc2
...
elif conditionN:     # not condition1 and ... and not conditionN-1 and conditionN
    blocN
else:              # not condition1 and ... and not conditionN
    blocE
suite
```

- Les conditons 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 = 'negative'
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, calculer son nombre de jours.

Formes équivalentes au **if**

Si et expressions booléennes

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

Ceci peut être réécrit avec une simple expression booléenne :

```
resultat = condition             |  resultat = not condition
```

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 structure de contrôle.
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 décroit 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
Question : Est-ce vraiment 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 : Nettoyage 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, et des chiffres.

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'occurence 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)

Répétion **for** (PourChaque)

Forme

```
for noms 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 un *itérable* (sera vu plus tard), par exemple une séquence (list, tuple, str...)
  - *noms* prend successivement chaque valeur de l'*itérable*
  - *bloc* est exécuté pour chaque affectation de *noms*
- **Principe :**
  - On sait combien de fois on exécute *bloc* (autant que d'éléments dans *expression*)
  - La terminaison est donc garantie... si l'*itérable* est fini.

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('{} -> {}'.format(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**.

Exceptions

Exceptions sur un exemple

Exemple

```
>>> 'bonjour'.index('j')
3
>>> 'bonjour'.index('z')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
```

L'index de 'j' est 3. Normal.
'z' n'est pas dans 'bonjour'. Que se passe-t-il ?

- `index` le signale en levant l'exception **ValueError** avec le message "substring not found"
- cette exception n'est jamais récupérée, l'interpréteur l'affiche avec la trace des appels.

Définition

Une exception est le moyen d'indiquer qu'une anomalie a été détectée, anomalie qui pourra être traitée par le code appelant.

Mécanisme d'exception

Principe

Mécanisme en trois phases :

1. Levée de l'exception quand une anomalie est détectée.
   C'est l'instruction **raise** qui le permet.
   L'exécution du bloc est interrompu et l'exception commence à se *propager*.
2. Propagation de l'exception : mécanisme automatique, l'exception remonte les blocs et les sous-programmes
3. Récupération de l'exception : la portion de code qui sait traiter l'exception *récupère* l'exception et l'exécution du code reprend dans le bloc associé à la récupération.

Intérêt

- découpler la partie du programme qui détecte une anomalie de la partie qui sait la traiter
  - plus pratique qu'un code d'erreur !
- permettre d'écrire un code optimiste (plus lisible) en regroupant après le traitement des erreurs

Lever une exception

Syntaxe sur un exemple

```
raise ValueError("J'ai levé ma première exception")
```

Remarque

La lever d'une exception se fait normalement dans un sous-programme.
C'est le moyen de signaler les éventuelles anomalies à l'appelant.

Récupérer une exception

Syntaxe

```
try:
    # lignes qui peuvent lever une exception
    # normalement au travers de l'appel de sous-programmes
except NomException1:
    # traitement de l'exception
    # On n'utilise pas l'objet exception récupéré
except NomException2 as nom:
    # traitement de l'exception
    # 'nom' contient l'objet exception qui se propageait
    ...
except BaseException as e: # on récupère presque toutes les exceptions !
    # traitement...
else:
    # Ne sera exécuté que si aucune exception n'est levée
finally:
    # Sera toujours exécuté, qu'il y ait une exception ou pas,
    # qu'elle soit récupérée ou pas
```

Explications

Dans les commentaires ci-dessus ;-)

Exemple complet

Le code

```
try:
    indice = chaine.index(sous_chaine)
except NameError as e:
    print('un nom est inconnu :', e)
except AttributeError as e:
    print("Certainement 'index' qui n'est pas trouvé :", e)
except TypeError as e:
    print("Certainement un problème sur le type de 'sous_chaine' :", e)
except ValueError as e:
    print("Certainement la sous-chaîne qui n'est pas trouvé :", e)
except Exception as e:
    print("C'est quoi celle là ?", e)
else:
    print("Super, pas d'exception levée.  L'indice est ", indice)
finally:
    print("Je m'exécute toujours !")
```

Les consignes

- Mettre ce code dans un fichier exemple_exception.py (par exemple) et l'exécuter.
- Ajouter en début `chaine = 10` et exécuter
- Changer `10` en `'bonjour'` et exécuter
- Ajouter `sous_chaine = 0` et exécuter
- Changer `0` en `'z'` et exécuter
- Changer `'z'` en `'j'` et exécuter

`with ... [as ...]`

Lire et afficher le contenu d'un fichier texte

```
nom_fichier = 'exemple.txt'
with open(nom_fichier, 'r') as fichier:
    for ligne in fichier:   # pour chaque ligne du fichier fichier
        print(ligne, end='')    # le '\n' est déjà dans `ligne`
```

with ... **as** est transformé en **try** ... **finallly**

```
try:
    fichier = None
    fichier = open(nom_fichier, "r")
    for ligne in fichier:
        print(ligne, end='')
finally:
    if fichier is not None:
        fichier.close()
```

Remarques

- `fichier.readline()` : retourne une nouvelle ligne du fichier (`''` quand plus de ligne à lire)
- `fichier.readlines()` : retourne la liste de toutes les lignes du fichier

Exemple : Écrire dans un fichier texte

Écrire dans un fichier

```
nom_fichier = '/tmp/exemple.txt'
with open(nom_fichier, 'w') as fichier:
    fichier.write('Première ligne\n')   # `write` n'ajoute pas de '\n'
    print('Deuxième ligne', file=fichier)
```

Remarque

- `write()` retourne le nombre de caractères effectivement écrits.

Utiliser des définitions d'autres modules

Utiliser des définitions d'autres modules

Définition

Un **module** est un fichier Python (extension .py) qui contient un ensemble de **définitions** qui peuvent être utilisées par d'autres modules (à condition de les importer).

Importer un module

```
import math, random # On donne simplement accès aux noms 'math' et 'random'

print(math.cos(math.pi / 3))    # définition préfixée du nom du module
print(random.randint(1, 100))
```

Importer les définitions d'un module

```
from math import pi, cos, pow   # accès à pi, cos et pow de math
from random import randint  # accès à randint de random

print(cos(pi / 3))      # pi directement accessible
print(randint(1, 100))
# math.pi ~~>  NameError: name 'math' is not defined
```

Exemple : les arguments de la ligne de commande avec le module sys

sys.argv : la liste des arguments de la ligne de commande.

Soit le fichier `exemple_argv.py` suivant :

exemple_argv.py

```
import sys

for arg in sys.argv:
    print("argument = '{}'".format(arg))
```

L'exécution `python3 exemple_argv.py premier 'un autre' ''` donne :

```
argument = 'exemple_argv.py'
argument = 'premier'
argument = 'un autre'
argument = ''
```

Remarque : voir le module os pour accéder aux fonctionnalités du système d'exploitation.

Exercices de synthèse

Jeu du devin

Le jeu du devin se joue à deux joueurs. Le premier joueur choisit un nombre compris entre 1 et 999. Le second doit le trouver en un minimum d'essais. À chaque proposition, le premier joueur indique si le nombre proposé est plus grand ou plus petit que le nombre à trouver. En fin de partie, le nombre d'essais est donné.

1. Écrire un programme dans lequel la machine choisit un nombre et le fait deviner à l'utilisateur.
2. Écrire un programme dans lequel l'utilisateur choisit un nombre et la machine doit le trouver. Pour chaque nombre proposé, l'utilisateur indique s'il est trop petit ('p' ou 'P'), trop grand ('g' ou 'G') ou trouvé ('t', 'T').
   On utilisera une recherche par dichotomie pour trouver le nombre.
3. Écrire le programme de jeu qui donne le choix à l'utilisateur entre deviner ou faire deviner le nombre.
4. Compléter le programme précédent pour que l'ordinateur propose de faire une nouvelle partie lorsque la précédente est terminée.

Word Count (wc)

Écrire un équivalent de la commande Unix wc (word count).
On n'implantera pas les options proposés par cette commande.
Exemple d'utilisation :

```
$ ls
fib.py
prime.py
ro-file
stats.py
tmp
wc.py
$ ./wc.py * zzz
     53     134    1103 fib.py
     32     112     871 prime.py
ro-file: Permission denied
     40     139     855 stats.py
tmp: Is a directory
     23     110     710 wc.py
zzz: No such file or directory
    148     495    3539 total
```

grep

Écrire un équivalent de la commande Unix grep.
Dans un premier temps, on ne proposera aucune option. Ensuite, on pourra ajouter le traitement des options `-h`, `-H` et `-v`.