Ressource R107
Fondamentaux de la programmation

IUT de Béziers, dépt. R&T © 2023
http://www.borelly.net/
Christophe.BORELLY@umontpellier.fr

Contenus de la ressource

- Notions d'algorithmique :
  - Variables, types de base (nombres, chaînes, listes/tableaux).
  - Structures de contrôle : tests, répétitions.
  - Fonctions et procédures.
  - Portée des variables.
- Tests et corrections d'un programme.
- Prise en main d'un environnement de programmation (éditeur, environnement de développement).
- Prise en main de bibliothèques, modules, d'objets existants (appels de méthodes).
- Manipulation de fichiers texte.
- Interaction avec le système d'exploitation et la ligne de commande : arguments,lancement de commandes.
- Suivi de versions (git, svn).

Généralités

- **python** est un langage de programmation orienté objet multi-plateforme avec entre autres :
  - un typage dynamique des variables
  - une gestion automatique de la mémoire
  - un système de gestion des erreurs (exceptions)
- Il peut s'utiliser directement dans l'interpréteur python ou bien à l'aide de fichiers texte : les scripts.

Généralités (2)

- Il existe principalement 2 versions (2.x et 3.x) qui ne sont pas totalement compatibles.
- Ce cours utilise la syntaxe du python 3.x.

```
$ python -V
Python 2.7.18
$ python3 -V
Python 3.9.6
$ python3
Python 3.9.6 (default, Jun 28 2021, 11:30:47)
[GCC 10.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello")
Hello
>>> exit
Use exit() or Ctrl-D (i.e. EOF) to exit
>>> exit()
```

Exemple de script

```
$ python3 hello.py
Hello world
$ chmod a+x hello.py ; ls -l
-rwxr-xr-x 1 cb users   68 aout  25 18:17 hello.py
$ ./hello.py
Hello world
```

- On peut ajouter au début du script, le « shebang » et le format de codage :

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
print("Hello world")
```

- NB : Les commentaires python commencent par #

Notation des nombres entiers

- Entiers en base 10 : ne commencent pas par `0`
- Entiers en base 16 : commencent par `0x`
- Entiers en base 8 : commencent par `0o`
- Entiers en base 2 : commencent par `0b`
- Les fonctions **`hex`**`()`, **`oct`**`()` et **`bin`**`()` affichent respectivement les valeurs en hexadécimal, octal et binaire.

```
>>> print(hex(160),0b10001011,bin(0x2C))
0xa0 139 0b101100
```

Nombres réels et complexes

- On peut utiliser la notation scientifique pour les nombres réels :

```
>>> 3.52e-3
0.00352
```

- On se sert de la lettre j (ou J) pour les nombres complexes :

```
>>> (3+4j)*(5+2j)
(7+26j)
>>> abs(3+2j)
3.605551275463989
>>> (3+2j).conjugate()
(3-2j)
```

Conversions entiers, réels et complexes

- Pour convertir en un nombre entier :

```
>>> int(4.5235e2)
452
>>> int('01100101',2) # En binaire
101
```

- Pour convertir en un nombre réel :

```
>>> float('+1E6')
1000000.0
```

- Pour fabriquer un nombre complexe :

```
>>> complex(4,5)
(4+5j)
```

Les opérateurs (1)

- Opérateurs :
  - `+` addition (3+2)
  - `-` soustraction (4-2)
  - `*` multiplication (5*2)
  - `**` puissance (3**2 => 9=3*3)
  - `@` multiplication de matrices
  - `/` division (5/2 => 2.5)
  - `//` division entière (17//3 => 5 car 17=3*5+2)
  - `%` reste de la division entière ou modulo (7%2 => 1)

Les opérateurs (2)

- Opérateurs d'affectation :
  - `=` affectation simple (x=5)
  - `+=` affectation et addition (x+=2 => x=x+2)
  - `-=` affectation et soustraction (x-=1 => x=x-1)
  - `*=` affectation et multiplication (x*=3 => x=x*3)
  - `**=` affectation et puissance (x**=2 => x=x**2)
  - `/=` affectation et division (x/=5 => x=x/5)
  - `//=` affectation et division entière (x//=2 => x=x//2)
  - `%=` affectation et modulo (x%=2 => x=x%2)

Les opérateurs (3)

- Opérateurs de comparaison:
  - **==** égalité
  - **!=** différence
  - **>** supérieur à
  - **>=** supérieur ou égal à
  - **<** inférieur à
  - **<=** inférieur ou égal à

Les opérateurs (4)

- Opérateurs logiques:
  - Valeurs booléennes : `True` et `False`
  - **`and`** => ET logique
    - x and y => Renvoi vrai si x et y sont vrais
  - **`or`** => OU logique
    - x or y => Renvoi vrai si x ou y sont vrais
  - **`not`** négation logique (ex. **`not`** `True`)
  - **`is`** et **`is not`** tests identité (ex. a **`is`** b)

Les opérateurs (5)

- Opérateurs sur les bits :
  - **<<** décalage à gauche (eq. multiplication par 2^x)
  - **>>** décalage à droite (eq. division par 2^x)
  - **&** => ET bits à bits (`0b1011 & 0b0110 => 2`)
  - **|** => OU bits à bits (`0b1011 | 0b0110 => 15`)
  - **^** => XOR bits à bits (`0b1011 ^ 0b0110 => 13`)
  - **~** inverse bit à bit (`x + ~x = -1`)

Les variables

- Le typage des variables est dynamique

```
>>> x=10
>>> x
10
>>> type(x)
<class 'int'>
>>> x=False
>>> type(x)
<class 'bool'>
```

```
>>> x=3.55e-3
>>> x
0.00355
>>> type(x)
<class 'float'>
>>> x="Hello"
>>> x
'Hello'
>>> type(x)
<class 'str'>
```

Les mots clés réservés

- Une variable commence par une lettre et peut contenir des chiffres et le caractère _.
- Les mots suivants ne peuvent pas être utilisés pour nommer des variables :

```
False      class      finally    is         return
None       continue   for        lambda     try
True       def        from       nonlocal   while
and        del        global     not        with
as         elif       if         or         yield
assert     else       import     pass
break      except     in         raise
```

Particularités de l’affectation

- S’il on veux avoir a=1, b=2 et c=3, il est possible d’écrire :

```
>>> a,b,c = 1,2,3
```

- Encore « plus fort » :

```
>>> a,b,c = b-4,c+2,a+3
>>> print(a,b,c)
-2 5 4
```

Documentation

- La fonction **`dir`**`()` permet de lister les méthodes utilisables sur n'importe quel objet.
- La fonction **`help`**`()` permet d'obtenir de l'aide sur une méthode donnée.
- Plus d'exemples et de précisions sur internet :
  - https://docs.python.org/3/
  - https://docs.python.org/3/library/
  - https://docs.python.org/3/reference/

Exemple sur les entiers

```
>>> dir(5)
['__abs__', '__add__', '__and__', '__class__', '__cmp__', '__coerce__', '__delattr__',
'__div__', '__divmod__', '__doc__', '__float__', '__floordiv__', '__format__',
'__getattribute__', '__getnewargs__', '__hash__', '__hex__', '__index__', '__init__',
'__int__', '__invert__', '__long__', '__lshift__', '__mod__', '__mul__', '__neg__',
'__new__', '__nonzero__', '__oct__', '__or__', '__pos__', '__pow__', '__radd__',
'__rand__', '__rdiv__', '__rdivmod__', '__reduce__', '__reduce_ex__', '__repr__',
'__rfloordiv__', '__rlshift__', '__rmod__', '__rmul__', '__ror__', '__rpow__',
'__rrshift__', '__rshift__', '__rsub__', '__rtruediv__', '__rxor__', '__setattr__',
'__sizeof__', '__str__', '__sub__', '__subclasshook__', '__truediv__', '__trunc__',
'__xor__', 'bit_length', 'conjugate', 'denominator', 'imag', 'numerator', 'real']
>>> help(int.bit_length)
bit_length(...)
int.bit_length() -> int
Number of bits necessary to represent self in binary.
>>> bin(37)
'0b100101'
>>> (37).bit_length()
6
```

Les chaînes de caractères

- La fonction **`str`**`(x)` permet de transformer **`x`** en chaîne de caractère.
- Sinon, on peut utiliser `'` ou `"` ou encore `'''` et `"""` pour définir une chaîne directement.
- Pour écrire sur plusieurs lignes, on utilise `\` à la fin de chaque ligne.
- Dans une chaîne, l'apostrophe s'écrit `\'` si le délimiteur est `'` et le guillemet `\"` si le délimiteur est `"`.
- `\n` passer à la ligne, `\t` tabulation, ...

```
>>> print('to"to\nti\'ti')
to"to
ti'ti
```

Les chaînes de caractères (2)

- La concaténation des chaînes se fait avec l'opérateur **+**.

```
>>> 'toro'+'toto'
'torototo'
```

- La classe **`str`** contient plusieurs méthodes intéressantes :

```
>>> dir('')
['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__',
'__getslice__', '__gt__', '__hash__', '__init__', '__le__', '__len__', '__lt__',
'__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '_formatter_field_name_split', '_formatter_parser',
'capitalize', 'center', 'count', 'decode', 'encode', 'endswith', 'expandtabs',
'find', 'format', 'index', 'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace',
'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'partition', 'replace',
'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines',
'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

Les chaînes de caractères (3)

```
>>> 'torototo'.count('to')
3
>>> 'torototo'.replace('to','XXX')
'XXXroXXXXXX'
>>> 'toro toto titi'.capitalize()
'Toro toto titi'
>>> 'toro toto titi'.title()
'Toro Toto Titi'
>>> 'toro toto titi'.upper()
'TORO TOTO TITI'
...
```

Formatage « old style » : Opérateur %

- On peut utiliser le formatage du langage C avec l'opérateur % sur une chaîne :

```
name,val='Bob',42
print('Hello, %s. val=%x !' % (name,val))
==>Hello, Bob. val=2a !

print('Hello, %(nom)s. val=%(v)x !' %
{'nom':name,'v':val})
==>Hello, Bob. val=2a !
```

Formatage « new style »

- La méthode `str.format()` utilise la même syntaxe que la classe `Formatter` (voir library : 6.1.3. Format String Syntax).
- Les valeurs remplacées dans le format sont entourée par des accolades `{}`.
- On peut inclure une accolade en la doublant : `{{` ou `}}`.

Exemples de formatages (2)

```
>>> '{0}, {1}, {2}'.format('a', 'b', 'c')
'a, b, c'
>>> '{2}, {1}, {0}'.format('a', 'b', 'c')
'c, b, a'
>>> '{}, {}, {}'.format('a', 'b', 'c')  # 3.1+ only
'a, b, c'
>>> 'Coord: {lat}, {long}'.format(lat='37.24N', long='-115.81W')
'Coord: 37.24N, -115.81W'
>>> "int: {0:d};  hex: {0:x};  oct: {0:o};  bin: {0:b}".format(42)
'int: 42;  hex: 2a;  oct: 52;  bin: 101010'
>>> "hex: {0:#x};  oct: {0:#o};  bin: {0:#b}".format(42)
'hex: 0x2a;  oct: 0o52;  bin: 0b101010'
>>> 'Correct answers: {0:.3f} or {0:.2%}'.format(1.0/3)
'Correct answers: 0.333 or 33.33%'
...
```

Les f-strings

- Depuis python 3.6 :

```
name,val='Bob',42
print(f'Hello, {name}. val={val} !')
==>Hello, Bob. val=42 !

print(f'Hello, {name}. val={val:#x} !')
==>Hello, Bob. val=0x2a !
```

Saisie au clavier

- En python 3 :

```
>>> x=input()
15
>>> x
'15'
>>> x=input("Entrer un entier : ")
Entrer un entier : 5
>>> x
'5'
```

- En python 2.7, utiliser `raw_input()`

Les structures de contrôle

- En python, on peut faire les structures de contrôles suivantes : `if`, `while`, `for`, `with` et `try`
- Les instructions `else` (sinon) sont optionnelles.
- Pour les boucles `while` et `for`, le mot clé `break` permet de « casser » la boucle. Si il y a un bloc `else`, il n'est pas exécuté.
- Le mot clé `continue` permet de sauter la fin de la boucle en retournant au début (test de l'expression).
- Le mot clé `pass` est une instruction sans effets permettant néanmoins de respecter la syntaxe des blocs d'instructions.
- On doit utiliser toujours la même indentation pour la définition des blocs d'instructions (eg. 2 ou 4 espaces, voire même 1 tabulation).

Les structures conditionnelles

```
x=2
if x>4:
   print('OK1')

if x<=3:
   print('OK2')
else:
   print('OK3')
```

```
if x%2==0:
   print('OK4')
   print('Pair')
elif x%3==1:
   print('OK5')
else:
   print('OK6')
```

OK2
OK4
Pair

Les boucles while

```
x=2
while x<5:
  print(x)
  x+=1
else:
  print('Fini1')
x=1
while x<10:
  if x%4==0:
    break
  print(x)
  x+=1
else:
  print('Fini2')
```

```
2
3
4
Fini1
1
2
3
```

```
print('Next loop')
x=1
while x<10:
  x+=1
  if x%4==0:
    continue
  print(x)
else:
  print('Fini3')
```

```
Next loop
2
3
5
6
7
9
10
Fini3
```

Les séquences

- En python, il y a 3 types de séquences de base possibles : les listes, les tuples et les ranges.
- On peut tester l'appartenance à une séquence avec les opérations `in` et `not in`.
- L'opérateur **+** permet de concaténer 2 séquences.
- L'opérateur ***** permet de dupliquer n fois une séquence.
- **`len`**`(s)` renvoi le nombre d'éléments
- **`max`**`(s)` et **`min`**`(s)` renvoient respectivement le plus grand et le plus petit élément.
- `s[i]` est l'élément i de la séquence (en commençant à 0)
- `s[i:j]` correspond aux éléments de i (inclus) à j (exclus)
- `s[i:j:k]` correspond aux éléments de i à j par pas de k

Les séquences et les chaînes (2)

- Les chaînes de caractères sont aussi des séquences, mais on ne peut pas les modifier.

```
>>> x='toto'
>>> min(x)
'o'
>>> max(x)
't'
>>> x[1:3]
'ot'
>>> x[1:4:2]
'oo'
```

Les listes

- Les listes sont des séquences modifiables et sont notées entre crochets **[ ]**.

```
>>> x=[] # Liste vide
>>> print('Taille:',len(x))
Taille : 0
>>> x=[2,3,5,7,11]
>>> print(x[1], x[1:3])
3 [3, 5]
>>> x.append(13)
>>> del x[3] # Supprime 7
>>> x.remove(11)
>>> print(x)
[2, 3, 5, 13]
>>> x.reverse()
>>> print(x)
[13, 5, 3, 2]
```

```
>>> y=x.pop() # Dépile
>>> print(x,y)
[13, 5, 3] 2
>>> z=x.pop(1)
>>> print(x,z)
[13, 3] 5
>>> x.insert(0,z)
>>> print(x)
[5, 13, 3]
>>> x.sort()
>>> print(x)
[3, 5, 13]
>>> x.sort(reverse=True)
>>> print(x)
[13, 5, 3]
```

Les tuples

- Les tuples ne sont pas modifiables et sont notés entre parenthèses **( )**.
- Lorsqu'il n'y a qu'un seul élément on ajoute une virgule finale : (1,)
- Les tuples implémentent toutes les opérations sur les séquences.

```
>>> x=(2,3,5,7,11)
>>> print(x[1], x[1:3])
3 (3, 5)
>>> print(max(x), min(x))
11 2
```

```
>>> y=tuple('abc')
>>> z=tuple([1,2,3])
>>> print(y,z)
('a', 'b', 'c') (1, 2, 3)
>>> yy=list(y)
>>> print(yy)
['a', 'b', 'c']
```

Les ranges

- Les ranges ne sont pas modifiables.
- Elles représentes une suite ordonnée de nombres et facilitent et accélèrent l'écriture de boucles `for`.
- Pour les visualiser, on peut les transformer en listes ou en tuples.

```
>>> print(list(range(5)))
[0, 1, 2, 3, 4]
>>> print(list(range(1,5)))
[1, 2, 3, 4]
>>> r=range(1,10,2)
>>> print(list(r))
[1, 3, 5, 7, 9]
```

```
>>> print(7 in r,8 in r,r.index(3))
True False 1
>>> print(r[2],r[-1])
5 9
```

Les boucles for

```
for x in [1,2,3]:
  print(x)
else:
  print('Fini0')

for x in (1,2,3):
  print(x)
else:
  print('Fini1')

for x in 'abcd':
  print(x)
else:
  print('Fini2')
```

```
1
2
3
Fini0
1
2
3
Fini1
a
b
c
d
Fini2
```

```
for x in range(5):
  print(x)
else:
  print('Fini3')

for x in range(1,5):
  print(x)
else:
  print('Fini4')

for x in range(5,1,-1):
  print(x)
else:
  print('Fini5')
```

```
0
1
2
3
4
Fini3
1
2
3
4
Fini4
5
4
3
2
Fini5
```

Listes en compréhension

- La notation `for in` peut être utilisée pour faciliter la création de listes et augmenter la lisibilité du code (ou pas) :

```
>>> [x**2 for x in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

Les sets

- Les sets correspondent à un ensemble non ordonnés de valeurs distinctes.
- Les valeurs sont définies entre accolades **`{}`**.
- Un set vide s'obtient avec **`set`**`()`.
- La version non modifiable s'obtient avec **`frozenset`**`()`.
- Comme les autres type de collections, les sets supportent les syntaxes : `len(set)`, `x in set`, `for x in set`.

Les dictionnaires

- Les dictionnaires permettent d'associer une clé à un objet.
- Ils sont définis avec des accolades `{}`.
- Il n'y pas de notion d'ordre comme dans une séquence.
- Ils possèdent entre autres, les méthodes **`keys`**`()` et **`values`**`()` qui renvoient des listes.
- La syntaxe « `key in dict` » remplace la méthode **`has_key`**`()` de python 2.7 qui permet de vérifier l'existence d'une clé donnée.
- La méthode **`get`**`(key,default=None)` peut être utilisée pour obtenir la valeur correspondante à la clé `key`. On peut aussi se servir de la syntaxe entre crochets naturellement.

Exemple de dictionnaire

```
dic={'one':3,'two':5,'three':7}

for key in dic:
  print(key,'=>',dic[key])
else:
  print('Fini1')

for key,val in dic.items():
  print(key,'=>',val)
else:
  print('Fini2')
```

```
one => 3
two => 5
three => 7
Fini1
one => 3
two => 5
three => 7
Fini2
```

Séquence d’octets

- La classe **`bytes`** permet de représenter une séquence non modifiable d’octets.
- On préfixe les objets de type `bytes` par la lettre b (ex. : `b'abc'`).

```
>>> x=bytes('toto','utf-8')
>>> print(x,x.decode())
b'toto' toto
>>> bytes.fromhex('414243 44 45')
b'ABCDE'
>>> b'\xf0\xf1\xf2'.hex()
'f0f1f2'
```

Autres séquences

- La classe **`bytearray`** produit par contre une séquence modifiable d'octets.
- On utilise `bytearray()` pour fabriquer les objets (ex. : `bytearray(b'abc')`).

```
>>> bytearray.fromhex('414243 44 45')
bytearray(b'ABCDE')
>>> bytearray(b'abc').hex()
'616263'
```

Les fonctions

- Le mot clé `def` permet de définir une fonction.
- La valeur de retour est indiquée si besoin, avec le mot clé `return`.
- On peut utiliser des arguments et leur donner une valeur par défaut.

```
>>> def f0(): print('Hello')
>>> def f1(x): return x**2
>>> def f2(x,y=5): return x+y
>>> f0()
Hello
>>> print(f1(2),f2(3),f2(4,2))
4 8 6
```

Appel de fonction

- On appelle généralement une fonction en précisant les valeurs des arguments dans l'ordre.

```
>>> def f1(x,y=2): return x+2*y
>>> f1(1)
5
>>> f1(1,3)
7
```

- Mais on peut aussi préciser le nom de l'argument lors de l'appel sans respecter forcément l'ordre.

```
>>> f1(y=4,x=2)
10
```

La fonction print()

```
print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout,
          flush=False)

    Prints the values to a stream, or to sys.stdout by default.

    Optional keyword arguments:
    sep:   string inserted between values, default a space.
    file:  a file-like object (stream);
           defaults to the current sys.stdout.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

Les fonctions anonymes

- Le mot clé `lambda` permet de définir des fonctions sans nom sur une seule ligne et avec une seule instruction.

```
>>> carre = lambda x : x**2
>>> [carre(i) for i in range(2,6)]
[4 9 16 25]
```

Utilisation de modules

- Les modules python additionnels doivent être ajoutées en début de script :

```
>>> import cbmod
>>> cbmod.f1(5)
25
>>> import cbmod as cb
>>> cb.f1(2)
4
>>> from cbmod import f1, f2
>>> f1(3)
9
>>> from cb.cbmod2 import *
>>> f3(7)
3.5
```

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
# Fichier cbmod.py
def f1(x): return x**2
def f2(x): return x+42
```

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
# Fichier cb/cbmod2.py
def f3(x): return x/2
def f4(x): return x*3
```

Le module operator

- Il est possible d'utiliser le module operator (voir library : 10.3.1. Mapping Operators to Functions) qui défini une fonction pour chaque opérateur.

```
>>> from operator import *
>>> mul(5,6)
30
>>> xor(3,6)
5
...
```

Le module math

- Le module math (voir library : 9.2. math - Mathematical functions) contient les principales fonctions mathématiques.

```
>>> from math import pi,sin,degrees,sqrt,log,gcd
>>> sin(pi/4)
0.7071067811865475
>>> degrees(pi/3)
59.99999999999999
>>> sqrt(2)    # Racine carrée
1.4142135623730951
>>> gcd(18,12) # Plus grand commun diviseur
6
```

Le module random

- Le module random (voir library : 9.6. random - Generate pseudo-random numbers) permet de générer simplement des nombres pseudo-aléatoires (non conseillé pour la cryptographie).

```
from random import randint,shuffle,getrandbits
x=[]
for i in range(10):
  x.append(randint(1,5))
print(x)

x=[randint(0,5) for i in range(10)]
print(x)
```

```
[2, 4, 2, 3, 1, 1, 5, 2, 2, 3]
[5, 0, 3, 4, 3, 2, 4, 1, 2, 0]
```

Le module random (2)

```
print('Mélange aléatoire:')
x = [i for i in range(10)]
shuffle(x)
print(x)

x=getrandbits(32)
print(bin(x))
```

```
Mélange aléatoire:
[5, 2, 1, 0, 4, 3, 9, 6, 8, 7]
0b10101110001101010100111000000110
```

Le module sys

- Le module sys (voir library : 29.1. sys — System-specific parameters and functions) contient plusieurs fonctions système et les flots `sys.stdin`, `sys.stdout` et `sys.stderr`.

```
>>> import sys
>>> sys.exit(42)
cb@pccb ~$ echo $?
42
cb@pccb ~$ ./cbargs.py toto titi
['./cbargs.py', 'toto', 'titi']
```

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
# Fichier cbargs.py
import sys
print(sys.argv)
```

Les arguments en python

- Il faut importer le module **sys**.
- On récupère des chaînes de caractère

```
import sys
print(f'Il y a {len(sys.argv)'} arguments')
print(sys.argv)
for a in sys.argv:
  print(f"Arg: {a}")
```

```
cb@pccb$ python3 cbargs.py 123 toto 456+
Il y a 4 arguments
['cbargs.py', '123', 'toto', '456+']
Arg: cbargs.py
Arg: 123
Arg: toto
Arg: 456+
```

Gestion des erreurs

- Certaines instructions peuvent générer une ou plusieurs erreurs appelées « exceptions » (voir library : 5.4. Exception hierarchy).
- On peut alors gérer ces erreurs dans un bloc `try : except`.
- Il peut y avoir plusieurs parties `except` pour un même `try`.
- Un même `except` peut correspondre à plusieurs types d'exceptions.
- On peut ajouter une clause `else` à un `try`. Il sera exécuté si on arrive normalement à la fin d'un try.
- La clause `finally` optionnelle sera exécuté dans tous les cas à la fin du `try : except`.
- Pour lancer une exception, on utilise le mot clé `raise`.

Gestion des erreurs

```
try:
   x = int(sys.argv[1])
except ValueError as err:
   print(err,file=sys.stderr)
except:
   pass
```

Les fichiers

- La fonction **`open`**`(file_name [, access_mode][, buffering])` permet d'ouvrir un fichier. Le mode d'accès est indiqué avec les lettres suivantes : `'r'` pour read, `'w'` pour write, `'a'` pour append, `'b'` pour mode binaire, et `'+'` pour activer les 2 modes read et write.
- La lecture peut se faire avec **`next`**`()`, **`read`**`([count])`, **`readline`**`([size])` ou **`readlines`**`([size])`.
  - NB : conserve les sauts de lignes.
- L'écriture peut se faire avec **`write`**`()` ou **`writelines`**`(seq)`.
  - NB : n'ajoute pas de sauts de ligne
- On peut utiliser **`tell`**`()` et **`seek`**`(offset[, from])` pour obtenir la position actuelle dans le fichier ou se déplacer dans celui-ci.

Exemple d’écriture puis de lecture dans un fichier

```
try:
  f=open('cb.txt','w')
  lines=[f'Ligne {i}\n' for i in range(5)]
  f.writelines(lines)
  f.close()
  with open('cb.txt','r') as f:
    for line in f:
      print(line.rstrip()) # Removes LF
except FileNotFoundError as err:
  print(err,file=sys.stderr)
```

```
Ligne 0
Ligne 1
Ligne 2
Ligne 3
Ligne 4
```

Les classes, objets et attributs

- Le mot clé `class` permet de définir une nouvelle classe d'objets.
- Les attributs (ou champs) sont les variables définies au premier niveau de la classe.
- On peut définir une rapide documentation qui sera visible avec l'attribut `__doc__`.
- Les méthodes sont des fonctions spécifiques aux objets. Elles possèdent au moins l'argument `self` qui permet d'interagir avec l'objet lui même.
- Le constructeur d'objets se défini avec `__init__()`.

Exemple (1)

```
class Personne:
  'Exemple de classe pour une personne'
  nom=None # Un attribut (ou champ)
  def __init__(self,n='toto'):
    self.nom=n
  # Appelé automatiquement lors
  # de la conversion en chaîne
  def __str__(self):
    return f"Personne({self.nom})"
  def disTonNom(self):
    print("Je m'appelle : ",self.nom)
```

Exemple (2)

```
>>> print(Personne.__doc__)
Exemple de classe pour une personne
>>> p=Personne()
>>> print(p)
Personne(toto)
>>> p.disTonNom()
Je m'appelle : toto
>>> p=Personne('alice')
>>> print(p)
Personne(alice)
>>> p.disTonNom()
Je m'appelle : alice
```

L’héritage

- L’héritage permet d’ajouter des fonctionnalités à un nouveau type d’objet basé sur une (ou plusieurs) classe(s) existante(nt).

```
class Francais(Personne):
  'Un francais est une personne'
  def __str__(self): # Surcharge
    return f"Francais({self.nom})"
  def disTonNom(self): # Surcharge
    print('Bonjour,')
    Personne.disTonNom(self)
```

```
>>> p=Francais('bob')
>>> print(p)
Francais(bob)
>>> p.disTonNom()
Bonjour,
Je m'appelle : bob
```

Les itérateurs

- Les itérateurs permettent de générer une séquence de valeurs.
- Il s'agit de classes définissant les méthodes **`__iter__`**( ) et **`__next__`**( ).
- L'exception **`StopIteration`** termine l'itération.

```
class CBImpairsIter:
  n=0
  def __init__(self,stop):
    self.stop=stop
  def __iter__(self):
    return self
  def __next__(self):
    self.n+=1
    if self.n>self.stop:
      raise StopIteration
    return 2*self.n-1
```

```
>>> [i for i in CBImpairsIter(10)]
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

Les générateurs

- Les générateurs sont plus compacts.
- L'instruction `yield` permet de renvoyer une valeur de l'itération.

```
def CBImpairs(n):
    for i in range(n):
        yield 2*i+1
```

```
>>> [i for i in CBImpairs(10)]
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

Les fonctions de base

```
abs()          dict()      help()       min()        setattr()
all()          dir()       hex()        next()       slice()
any()          divmod()    id()         object()     sorted()
ascii()        enumerate() input()      oct()      staticmethod()
bin()          eval()      int()        open()       str()
bool()         exec()      isinstance() ord()        sum()
bytearray()    filter()    issubclass() pow()        super()
bytes()        float()     iter()       print()      tuple()
callable()     format()    len()        property()   type()
chr()          frozenset() list()       range()      vars()
classmethod()  getattr()   locals()     repr()       zip()
compile()      globals()   map()        reversed()   __import__()
complex()      hasattr()   max()        round()
delattr()      hash()      memoryview() set()
```

Les fonctions de base (2)

- **`enumerate`**`()` renvoi un itérateur contenant des tuples avec un indexe.

```
>>> list(enumerate('abc'))
[(0, 'a'), (1, 'b'), (2, 'c')]
>>> list(enumerate('abc',11))
[(11, 'a'), (12, 'b'), (13, 'c')]
```

- **`reversed`**`()` renvoi un itérateur en partant de la fin.

```
>>> ''.join(reversed('abc'))
'cba'
```

La fonction map

- Permet d'exécuter une fonction sur chaque élément d'une séquence :
- Renvoi un iterable.

```
>>> carre = lambda x : x**2
>>> list(map(carre,range(5)))
[0 1 4 9 16]
```

La fonction zip()

- Cette fonction permet de créer un itérateur de tuples avec l'agrégation de chaque éléments

```
>>> list(zip('ABCD','xy'))
[('A', 'x'), ('B', 'y')]
```

La fonction sorted()

- Cette fonction renvoi un itérateur ordonné de l'*iterable* fourni :
  - **sorted**(iterable,key=None,reverse=False)

```
>>> ''.join(sorted('BaDc'))
'BDac'
>>> ''.join(sorted('BaDc',reverse=True))
'caDB'
>>> ''.join(sorted('BaDc',key=str.lower))
'aBcD'
```

Trier un dictionnaire

```
x={'x':15,'h':1,'o':23,'u':-5}
# tri par valeur
y={k:v for k,v in sorted(x.items(),key=lambda kv:kv[1])}
print(y)
#{'u': -5, 'h': 1, 'x': 15, 'o': 23}

# tri par clé
z={k:v for k,v in sorted(x.items(),key=lambda kv:kv[0])}
print(z)
#{'h': 1, 'o': 23, 'u': -5, 'x': 15}
```

Références

- https://www.python.org/
- https://docs.python.org/