Programmation en Python

Initiation

Pierre Pompidor

Introduction - Python vs les autres langages de script

Python est un langage ubiquiste pour

- le prototypage et la création d'applications
- l'extension d'applications
- le calcul scientifique
- le scripting système
- la fouille de données (data mining), ...

Introduction - Python vs les autres langages de script

Python est un langage ubiquiste pour

- le prototypage et la création d'applications
- l'extension d'applications
- le calcul scientifique
- le scripting système
- la fouille de données (data mining), ...

Python est un langage multiplateformes

Introduction - Python vs les autres langages de script

Python est un langage ubiquiste pour

- le prototypage et la création d'applications
- l'extension d'applications
- le calcul scientifique
- le scripting système
- la fouille de données (data mining), ...

Python est un langage multiplateformes

Python est un langage modulaire

Introduction - Python vs les autres langages de script

Python est un langage ubiquiste pour

- le prototypage et la création d'applications
- l'extension d'applications
- le calcul scientifique
- le scripting système
- la fouille de données (data mining), ...

Python est un langage multiplateformes

Python est un langage modulaire

Python est un langage efficace

Introduction - Python vs les autres langages de script

Python est un langage ubiquiste pour

- le prototypage et la création d'applications
- l'extension d'applications
- le calcul scientifique
- le scripting système
- la fouille de données (data mining), ...

Python est un langage multiplateformes

Python est un langage modulaire

Python est un langage efficace

Python est un langage pédagogique

Introduction - Python est un langage interprété

L'interpréteur python : python3

- traduit, une par une, chaque instruction en langage binaire
- il n'y a pas d'exécutable stocké sur votre système de fichiers (contrairement par exemple à un programme écrit avec le langage C ou C++)

Introduction - Python est un langage interprété

L'interpréteur python : python3

- traduit, une par une, chaque instruction en langage binaire
- il n'y a pas d'exécutable stocké sur votre système de fichiers (contrairement par exemple à un programme écrit avec le langage C ou C++)

Il est possible de programmer

- directement sous l'intepréteur (python3) :
  python3
  >>> ...
  exit()
- en créant des scripts (d'extensions **.py**)
  exécutés dans un terminal / environnement de développement
  python3 bonjour.py

Introduction - Python est modulaire

Notion de modules

- Un module est un fichier contenant du code python
- Un module permet de ne charger que le minimum en mémoire
- Ainsi du code peut-être réemployé dans une autre application

Introduction - Python est modulaire

Notion de modules

- Un module est un fichier contenant du code python
- Un module permet de ne charger que le minimum en mémoire
- Ainsi du code peut-être réemployé dans une autre application

Exemples d'utilisation de modules

```
import numpy
from math import sqrt
```

Introduction - Python est modulaire

Notion de modules

- Un module est un fichier contenant du code python
- Un module permet de ne charger que le minimum en mémoire
- Ainsi du code peut-être réemployé dans une autre application

Exemples d'utilisation de modules

```
import numpy
from math import sqrt
```

Installation d'un module "public"

```
pip3 install <nomModule>
```

Introduction - Python est modulaire

Notion de modules

- Un module est un fichier contenant du code python
- Un module permet de ne charger que le minimum en mémoire
- Ainsi du code peut-être réemployé dans une autre application

Exemples d'utilisation de modules

```
import numpy
from math import sqrt
```

Installation d'un module "public"

pip3 install <nomModule>

Bibliothèque standard Python : modules en Python et en C

Introduction - Python est modulaire

Notion de modules

- Un module est un fichier contenant du code python
- Un module permet de ne charger que le minimum en mémoire
- Ainsi du code peut-être réemployé dans une autre application

Exemples d'utilisation de modules

```
import numpy
from math import sqrt
```

Installation d'un module "public"

pip3 install <nomModule>

Bibliothèque standard Python : modules en Python et en C

Frameworks Python non mis en œuvre dans ce cours

Introduction - Un premier exemple de script python

bonjour.py

```
nom = input("Quel␣est␣ton␣nom␣?␣")
print("Bonjour", nom)
```

Introduction - Un premier exemple de script python

bonjour.py

```
nom = input("Quel␣est␣ton␣nom␣?␣")
print("Bonjour", nom)
```

Exécution dans un shell (terminal Linux) via l'interpréteur

python3 bonjour.py

Introduction - Un premier exemple de script python

bonjour.py

```
nom = input("Quel est ton nom ? ")
print("Bonjour", nom)
```

Exécution dans un shell (terminal Linux) via l'interpréteur

python3 bonjour.py

Exécution directe dans un shell

La première ligne du script doit être : #!/usr/bin/env python3
Il faut donner le droit d'exécution au script : chmod +x bonjour.py
./bonjour.py

Exécution via un IDE (environnement de développement)

Par exemple : Control-F5 sous Visual Studio Code

Entrées/sorties - print()

La fonction print() permet d'utiliser un format

```
print(f"Je suis {prenom} {nom}")

print("Je suis {} {}".format(prenom, nom))

print("Je suis %s %s"%(prenom, nom))
```

Entrées/sorties - print()

La fonction print() permet d'utiliser un format

```
print(f"Je suis {prenom} {nom}")

print("Je suis {} {}".format(prenom, nom))

print("Je suis %s %s"%(prenom, nom))
```

Le format permet de définir des formats d'affichage ;). Exemples avec % :

- `%2d`
- `%3.2f`

Comment développer en Python

Utilisation d'un éditeur de texte "générique"

Les classiques : *vi*, *nano*, *atom*, *sublime text*, ... et **emacs**

Des environnements de développement comme :

- Une bonne solution : **Visual Studio Code** (libre)
  code
- **pyCharm** : `https://www.jetbrains.com/pycharm/`
- **pyzo** : `https://pyzo/`
- **spyder** : `https://www.spyder-ide.org/`

Des environnements de développement en ligne tels que :

- **jupyter** : `https://jupyter.org/`
- **cocalc** : `https://cocalc.com/`

Structures de données de base : Python permet de créer

des variables scalaires qui ne contiennent qu'une seule valeur (notamment numériques)

Structures de données de base : Python permet de créer

des variables scalaires qui ne contiennent qu'une seule valeur (notamment numériques)

des chaînes de caractères

Structures de données de base : Python permet de créer

des variables scalaires qui ne contiennent qu'une seule valeur (notamment numériques)

des chaînes de caractères

des variables composites qui regroupent plusieurs valeurs :

- les **listes** : tableaux dynamiques
- les **tuples** : éléments non directement modifiables
- les **dictionnaires** : éléments indexés par une clef
  accès direct

Structures de données de base : Python permet de créer

des variables scalaires qui ne contiennent qu'une seule valeur (notamment numériques)

des chaînes de caractères

des variables composites qui regroupent plusieurs valeurs :

- les **listes** : tableaux dynamiques
- les **tuples** : éléments non directement modifiables
- les **dictionnaires** : éléments indexés par une clef
  accès direct

des **objets** (paradigme de la **programmation par objets**)

incarnent des concepts et regroupent plusieurs variables et les traitements qui peuvent leurs être appliqués

Structures de données de base

Une variable correspond à l'allocation d'une zone de la mémoire vive

Connaître le type d'une variable

```
i = 1
print(type(i))   # affiche <class 'int'>
```

Structures de données de base

Une variable correspond à l'allocation d'une zone de la mémoire vive

Connaître le type d'une variable

```
i = 1
print(type(i))   # affiche <class 'int'>
```

Les variables scalaires

```
i = 1
f = 1.0
booleen = False
```

Structures de données de base

Une variable correspond à l'allocation d'une zone de la mémoire vive

Connaître le type d'une variable

```
i = 1
print(type(i))   # affiche <class 'int'>
```

Les variables scalaires

```
i = 1
f = 1.0
booleen = False
```

Le typage est dynamique

Le type de la variable n'est pas précisé lors de sa création

Structures de données de base (suite)

Allocation/désallocation automatique de la mémoire

- La mémoire nécessaire est automatiquement allouée
- Et cela permet d'éviter le principal risque de bug

Structures de données de base (suite)

Allocation/désallocation automatique de la mémoire

- La mémoire nécessaire est automatiquement allouée
- Et cela permet d'éviter le principal risque de bug

Erreur fréquente en C

```
char* chaine = "Bonjour";
// il y a deja un risque
strcat(chaine, " toi"); // second risque
   // (meme si cela semble fonctionner)
```

Structures de données de base (suite)

Allocation/désallocation automatique de la mémoire

- La mémoire nécessaire est automatiquement allouée
- Et cela permet d'éviter le principal risque de bug

Erreur fréquente en C

```
char* chaine = "Bonjour";
// il y a deja un risque
strcat(chaine, " toi"); // second risque
   // (meme si cela semble fonctionner)
```

Pas de souci en python

```
chaine = "Bonjour"
chaine += " toi"
```

Structures de données de base - les chaînes (strings)

Exemple de création de chaînes de caractères

```
chaine1 = "Bonjour"
chaine2 = 'Hello'
chaine3 = "Aujourd'hui"
chaineVide = ""
autreExempleDeVacuite = ''
```

Structures de données de base - les chaînes (strings)

Exemple de création de chaînes de caractères

```
chaine1 = "Bonjour"
chaine2 = 'Hello'
chaine3 = "Aujourd'hui"
chaineVide = ""
autreExempleDeVacuite = ''
```

Nombre de caractères d'une chaîne : len()

```
print(len(chaine3))   # Affiche : 11
```

Structures de données - les listes

Les listes sont des tableaux dynamiques

```
menu = ['entree', 'plat', 'dessert']
print(menu[1])   # affiche : plat
```

Structures de données - les listes

Les listes sont des tableaux dynamiques

```
menu = ['entree', 'plat', 'dessert']
print(menu[1])   # affiche : plat
```

Syntaxe

- Les crochets encadrent les éléments de la liste
- Les éléments sont indicés par un entier (0 pour le pr. élément)
- Un élément d'une liste peut être de n'importe quelle valeur

Structures de données - les listes

Les listes sont des tableaux dynamiques

```
menu = ['entree', 'plat', 'dessert']
print(menu[1])   # affiche : plat
```

Syntaxe

- Les crochets encadrent les éléments de la liste
- Les éléments sont indicés par un entier (0 pour le pr. élément)
- Un élément d'une liste peut être de n'importe quelle valeur

Une liste peut-être initialisée à vide

```
listeVide = []
```

Structures de données - opérations sur les listes

liste = [1, 2, 3]

```
liste.append("partez␣!")   # ajoute un element
del liste[2]               # supprime le 3ieme element
liste.remove(1)            # supprime le premier element
                           # qui a pour valeur 1
```

Structures de données - opérations sur les listes

liste = [1, 2, 3]

```
liste.append("partez␣!")   # ajoute un element
del liste[2]               # supprime le 3ieme element
liste.remove(1)            # supprime le premier element
                           # qui a pour valeur 1
```

Les fonctions sont de différentes "espèces"

- append() et remove() : **méthodes** sur l'**objet** liste
- del est une pseudo-fonction

Structures de données - opérations sur les listes

liste = [1, 2, 3]

```
liste.append("partez␣!")   # ajoute un element
del liste[2]               # supprime le 3ieme element
liste.remove(1)            # supprime le premier element
                           # qui a pour valeur 1
```

Les fonctions sont de différentes "espèces"

- append() et remove() : **méthodes** sur l'**objet** liste
- del est une pseudo-fonction

Test d'appartenance d'un élement à une liste :

```
if nom in listeEleves :
    print("Ah␣celui-la,␣je␣le␣connais␣bien")
```

Structures de données - les listes (suite)

Les listes peuvent elles-mêmes contenir des listes (ou d'autres structures de données)

```
liste = [1, 2, 3, ["partez", "maintenant"]]
print(liste[3][1])   # affiche : maintenant
```

Indiçage négatif

```
menu = ["Attention", 1, 2, 3, "partez"]
print(menu[-2])   # affiche : 3
```

Structures de données - les listes (suite)

Les listes peuvent elles-mêmes contenir des listes (ou d'autres structures de données)

```
liste = [1, 2, 3, ["partez", "maintenant"]]
print(liste[3][1])   # affiche : maintenant
```

Indiçage négatif

```
menu = ["Attention", 1, 2, 3, "partez"]
print(menu[-2])   # affiche : 3
```

Tranches de listes (slices)

```
liste = [1, 2, 3, "partez", "!"]
print(liste[1:4])   # [1,4[
                    # affiche [2, 3, "partez"]
```

Un exemple de liste : la liste des paramètres du script

Il faut importer le module sys

Un exemple de liste : la liste des paramètres du script

Il faut importer le module sys

La liste des paramètres est sys.argv

Un exemple de liste : la liste des paramètres du script

Il faut importer le module sys

La liste des paramètres est sys.argv

Soit le script bonjour.py

```
import sys

print(sys.argv[0])
print("Bonjour", sys.argv[1])
```

Un exemple de liste : la liste des paramètres du script

Il faut importer le module sys

La liste des paramètres est sys.argv

Soit le script bonjour.py

```
import sys

print(sys.argv[0])
print("Bonjour", sys.argv[1])
```

./bonjour.py Pierre

```
./bonjour.py
Bonjour Pierre
```

Structures de données - les listes vs. les tuples

Les tuples : des séquences constantes (presque non modifiables)

- Les tuples sont encadrés par des parenthèses
- Mais l'accès aux éléments se fait toujours via les crochets

Structures de données - les listes vs. les tuples

Les tuples : des séquences constantes (presque non modifiables)

- Les tuples sont encadrés par des parenthèses
- Mais l'accès aux éléments se fait toujours via les crochets

Un exemple de tuple

```
unTuple = (1, 2, 3)
print(unTuple[0])
unTuple[0] = "un"   # cela provoque une erreur !
```

Structures de données - les listes vs. les tuples

Les tuples : des séquences constantes (presque non modifiables)

- Les tuples sont encadrés par des parenthèses
- Mais l'accès aux éléments se fait toujours via les crochets

Un exemple de tuple

```
unTuple = (1, 2, 3)
print(unTuple[0])
unTuple[0] = "un"   # cela provoque une erreur !
```

Une variable composite en valeur d'un tuple

peut être modifiée...

Structures de données de base - les listes (suite)

Copie de listes

```
liste = [1, 2, 3, "partez", "!"]
nouvelleListe = liste[:]
```

Structures de données de base - les listes (suite)

Copie de listes

```
liste = [1, 2, 3, "partez", "!"]
nouvelleListe = liste[:]
```

Une copie intuitive qui ne fonctionne pas :

```
liste = [1, 2, 3, "partez", "!"]
nouvelleListe = liste
```

Structures de données de base - les listes (suite)

Copie de listes

```
liste = [1, 2, 3, "partez", "!"]
nouvelleListe = liste[:]
```

Une copie intuitive qui ne fonctionne pas :

```
liste = [1, 2, 3, "partez", "!"]
nouvelleListe = liste
```

Utilisez *id()*

pour connaître l'emplacement des variables en mémoire

Structures de données de base - les listes (suite)

Copie de listes

```
liste = [1, 2, 3, "partez", "!"]
nouvelleListe = liste[:]
```

Une copie intuitive qui ne fonctionne pas :

```
liste = [1, 2, 3, "partez", "!"]
nouvelleListe = liste
```

Utilisez *id()*

pour connaître l'emplacement des variables en mémoire

Copie de listes en profondeur

```
nouvelleListe = copy.deepcopy(liste)
```

Structures de données de base

Les chaînes de caractères ne sont pas des listes

```
chaine = "sybillin"
listeDeCaracteres = list(chaine)
listeDeCaracteres[1] = 'i'
listeDeCaracteres[3] = 'y'
print(listeDeCaracteres)
chaine = "".join(listeDeCaracteres)
print(chaine)
```

Structures de données de base

Les chaînes de caractères ne sont pas des listes

```
chaine = "sybillin"
listeDeCaracteres = list(chaine)
listeDeCaracteres[1] = 'i'
listeDeCaracteres[3] = 'y'
print(listeDeCaracteres)
chaine = "".join(listeDeCaracteres)
print(chaine)
```

list() crée une liste

join() est une méthode sur une chaîne de caractères

Une chaîne de caractères vide est ici utilisée pour l'appliquer

Structures de base - le bloc d'instructions

Exemple de bloc d'instruction assujetti à un test

```
if <expression conditionnelle> :
    instruction 1
    instruction 2
    ...
    instruction n
instruction hors du bloc (car alignee sur le if)
```

Structures de base - le bloc d'instructions

Exemple de bloc d'instruction assujetti à un test

```
if <expression conditionnelle> :
    instruction 1
    instruction 2
    ...
    instruction n
instruction hors du bloc (car alignee sur le if)
```

Syntaxe d'un bloc d'instructions

- Le bloc d'instructions est initié par un :
- Les instructions sont indentées
  ne mélangez pas des espaces avec des indentations

Structures de base - La structure conditionnelle

Exemple de structure conditionnelle :

```
import sys
if len(sys.argv) != 2 :
    print("Le␣script␣doit␣avoir␣un␣parametre")
    exit()
```

Structures de base - La structure conditionnelle

Exemple de structure conditionnelle :

```
import sys
if len(sys.argv) != 2 :
    print("Le␣script␣doit␣avoir␣un␣parametre")
    exit()
```

Le "sinon" est introduit par else

```
if <expression conditionnelle> :
    premiere instruction du bloc
    ...
else :
    premiere instruction du bloc
    ...
```

Structures de base - La structure conditionnelle

Exemple de structures conditionnelle

```
if (i > 10 and j < 10) or (i < 10 and j > 10) :
    print(i, "et", j, "ne se comprennent pas")
```

Structures de base - La structure conditionnelle

Exemple de structures conditionnelle

```
if (i > 10 and j < 10) or (i < 10 and j > 10) :
    print(i, "et", j, "ne se comprennent pas")
```

Opérateurs

- opérateurs logiques : and or
- opérateurs de comparaison classiques
  attention : == et !=
- une expression renvoie faux si l'évaluation est égale à 0 ou ""

Exemple d'une boucle while avec indice de boucle

```
liste = [1, 2, 3, "partez", "!"]
i = 0
while i < len(liste) :
        print(liste[i])
        i += 1
```

Structures de base - Les structures itératives

Exemple d'une boucle while avec indice de boucle

```
liste = [1, 2, 3, "partez", "!"]
i = 0
while i < len(liste) :
        print(liste[i])
        i += 1
```

Syntaxe

```
<initialisation eventuelle d'un indice de boucle>
while <expression conditionnelle> :
      instructions du bloc
      <(in|de)crement eventuel d'un indice>
```

Exemple d'une boucle for automatique

```
liste = [1, 2, 3, "partez", "!"]
for valeur in liste :
    print(valeur)
```

Structures de base - Les structures de boucle

Exemple d'une boucle for automatique

```
liste = [1, 2, 3, "partez", "!"]
for valeur in liste :
    print(valeur)
```

Syntaxe

```
for <variable locale> in <iterable> :
    instructions du bloc
```

Les compréhensions de listes

Une syntaxe compacte pour initialiser une liste

Exemple de compréhension de liste

```
liste = [i*2 for i in range(10)]
for e in liste :
        print(e)
```

Les compréhensions de listes

Une syntaxe compacte pour initialiser une liste

Exemple de compréhension de liste

```
liste = [i*2 for i in range(10)]
for e in liste :
        print(e)
```

Rappel sur la fonction *range()*

- range() permet de générer une suite d'entiers :
  range(10) génère les entiers de 0 à 9

Un premier exemple "synthétique"

Salutation de tous les paramètres donnés au script

```
# -*- coding: utf-8 -*-
import sys

if len(sys.argv) > 1 :
    for parametre in sys.argv[1:] :
        print("Bonjour␣"+parametre)
else :
    print("J'aimerais␣tellement␣etre␣poli␣!")
```

Un slice est utilisé

pour éluder le nom du script

Structures de données - Les dictionnaires

Exemple de dictionnaire

```
nbJoursMois = {'janvier':31, 'fevrier':28.25 }
print(nbJoursMois['fevrier'])
# Affiche : 28.25
```

Structures de données - Les dictionnaires

Exemple de dictionnaire

```
nbJoursMois = {'janvier':31, 'fevrier':28.25 }
print(nbJoursMois['fevrier'])
# Affiche : 28.25
```

Les dictionnaires permettent :

- de rajouter de la sémantique (les éléments sont nommés par une clef)
- d'accéder directement à l'emplacement de l'élément (généralement via une fonction de hashage)

Structures de données - Les dictionnaires

Exemple de dictionnaire

```
nbJoursMois = {'janvier':31, 'fevrier':28.25 }
print(nbJoursMois['fevrier'])
# Affiche : 28.25
```

Les dictionnaires permettent :

- de rajouter de la sémantique (les éléments sont nommés par une clef)
- d'accéder directement à l'emplacement de l'élément (généralement via une fonction de hashage)

Un dictionnaire peut-être initialisé à vide

```
dicoVide = {}
```

Structures de données - Les dictionnaires (suite)

Affichage des éléments d'un dictionnaire

```
nbJoursMois = {'janvier':31, 'fevrier':28.25,
               'mars':31, ...}
for clef in nbJoursMois :
    print(clef, ":", nbJoursMois[clef])

for (clef, valeur) in nbJoursMois.items() :
    print(clef, ":", valeur)
```

Structures de données - Les dictionnaires (suite)

Affichage des éléments d'un dictionnaire

```
nbJoursMois = {'janvier':31, 'fevrier':28.25,
               'mars':31, ...}
for clef in nbJoursMois :
    print(clef, ":", nbJoursMois[clef])

for (clef, valeur) in nbJoursMois.items() :
    print(clef, ":", valeur)
```

L'ordre d'affichage des éléments

- ne correspond pas à l'ordre de création de ces éléments
- mais il est possible d'utiliser des dictionnaires ordonnés

Structures de données - Les dictionnaires (suite)

Exemple de compréhension de dictionnaire

```
lettres = "abcdefghijklmnopqrstuvwxyz"
dico = {lettres[i-1]:i for i in range(1, 27)}
for (k, v) in dico.items() :
    print(k, v)
```

Structures de données - Les dictionnaires (suite)

Exemple de compréhension de dictionnaire

```
lettres = "abcdefghijklmnopqrstuvwxyz"
dico = {lettres[i-1]:i for i in range(1, 27)}
for (k, v) in dico.items() :
    print(k, v)
```

Le dictionnaire a pour clef chaque lettre de l'alphabet

associée en valeur à son ordre; par exemple :
1 en valeur de `dico['a']`

Structures de données - Les dictionnaires (suite)

Mise à jour d'un dictionnaire

Tester si une potentielle nouvelle clef existe déjà
Sinon ajouter directement l'élément en le nommant par sa clef

Structures de données - Les dictionnaires (suite)

Mise à jour d'un dictionnaire

Tester si une potentielle nouvelle clef existe déjà
Sinon ajouter directement l'élément en le nommant par sa clef

Exemple d'implémentation (avec incrémentation si la clef existe déjà)

```
if clef in dictionnaire :
    dictionnaire[clef] += 1
else :
    dictionnaire[clef] = 1
```

Fonctions

Une fonction est un bloc d'instructions paramétré

```
def somme(a, b) :
    return a+b

print(somme(1,2))
```

Fonctions

Une fonction est un bloc d'instructions paramétré

```
def somme(a, b) :
    return a+b

print(somme(1,2))
```

Une fonction permet de factoriser du code

Fonctions

Une fonction est un bloc d'instructions paramétré

```
def somme(a, b) :
    return a+b

print(somme(1,2))
```

Une fonction permet de factoriser du code

Des fonctions particulières

- Une fonction peut ne pas renvoyer de résultat (procédure)
- Une fonction peut elle-même se rappeler (fonction récursive)

Fonctions (suite)

Exemple de squelette d'une fonction récursive

```
def decompte (n) :
    print(n)
    if n == 0 :
        print("FIN␣!")
        return
    decompte(n-1)

decompte(10)
```

Fonctions (suite)

Exemple de squelette d'une fonction récursive

```
def decompte (n) :
    print(n)
    if n == 0 :
        print("FIN␣!")
        return
    decompte(n-1)

decompte(10)
```

Question existentielle

Comment une variable (ici n) peut-elle exister plusieurs fois en même temps ?

Fonctions récursives (suite)

La **pile**

est un espace de la mémoire réservée au programme
dans laquelle sont mémorisées les variables locales à une fonction :
- ses paramètres
- les variables créées dans la fonction
=> c'est le **contexte d'exécution** de la fonction

Fonctions récursives (suite)

La **pile**

est un espace de la mémoire réservée au programme
dans laquelle sont mémorisées les variables locales à une fonction :
- ses paramètres
- les variables créées dans la fonction
=> c'est le **contexte d'exécution** de la fonction

Dans le cas d'une fonction récursive

plusieurs contextes d'exécutions de cette fonction y sont stockés
lors du réappel de la fonction (descente de récursivité)
puis progressivement désalloués en remontée de récursivité

Fonctions récursives (suite)

Factorielle en récursif

```
import sys

def factorielle(n) :
    if n == 0 :
        return 1
    return n * factorielle(n-1)

print(factorielle(sys.argv[1]))
```

Fonctions récursives (suite)

Factorielle en récursif

```
import sys

def factorielle(n) :
    if n == 0 :
        return 1
    return n * factorielle(n-1)

print(factorielle(sys.argv[1]))
```

Construction du résultat

en *remontée de la récursivité !*

Intérêt de la récursivité

Parcours (et création) d'**arbres**

explorateur de dossiers, IA de jeu...

Intérêt de la récursivité

Parcours (et création) d'**arbres**

explorateur de dossiers, IA de jeu...

Parcours (et création) de **graphes**

plus court chemin, optimisation de flux...

Intérêt de la récursivité

Parcours (et création) d'**arbres**

explorateur de dossiers, IA de jeu...

Parcours (et création) de **graphes**

plus court chemin, optimisation de flux...

Programmation d'algorithmes

reposant sur des structures de données complexes

Intérêt de la récursivité

Parcours (et création) d'**arbres**

explorateur de dossiers, IA de jeu...

Parcours (et création) de **graphes**

plus court chemin, optimisation de flux...

Programmation d'algorithmes

reposant sur des structures de données complexes

Efficacité

en cas de double implémentation possible (itérative et récursive)
souvent l'implémentation itérative est plus efficace
(mais l'implémentation récursive est plus élégante)

Fonctions (suite)

Paramètres variables avec *args (args est une convention)

```
def fonction(*args) :
    for arg in args :
        print("Bonjour", arg)
fonction("Pierre", "Paul")
```

Fonctions (suite)

Paramètres variables avec *args (args est une convention)

```
def fonction(*args) :
    for arg in args :
        print("Bonjour", arg)
fonction("Pierre", "Paul")
```

Paramètres variables sous formes de clefs/valeurs avec **kvargs

```
def fonction(**kwargs) :
    for (k, v) in kwargs.items() :
        print(k, v)

fonction(merveille1="Pyramides␣de␣Kheops",
         merveille2="Jardins␣suspendus␣...")
```

Fonctions (suite)

Des fonctions peuvent être regroupées dans un module (ici monModule.py)

```
def testModule() :
    print("Je␣suis␣dans␣testModule()")
```

Fonctions (suite)

Des fonctions peuvent être regroupées dans un module (ici monModule.py)

```
def testModule() :
    print("Je␣suis␣dans␣testModule()")
```

Mise en oeuvre du module monModule.py

```
#!/usr/bin/env python3

import monModule

monModule.testModule()
```

Fonctions (suite)

Des fonctions peuvent être regroupées dans un module (ici monModule.py)

```
def testModule() :
    print("Je␣suis␣dans␣testModule()")
```

Mise en oeuvre du module monModule.py

```
#!/usr/bin/env python3

import monModule

monModule.testModule()
```

Le nom du module fait office d'espace de noms

Des fonctions qui n'en sont pas : les générateurs

Un exemple de générateur qui décompte à partir de 10

```
def creeGenerateur() :
    for i in range(10, 0, -1) :
        yield i

generateur = creeGenerateur()
print(next(generateur))   # affiche : 10
print(next(generateur))   # affiche : 9
```

Des fonctions qui n'en sont pas : les générateurs

Un exemple de générateur qui décompte à partir de 10

```
def creeGenerateur() :
    for i in range(10, 0, -1) :
        yield i

generateur = creeGenerateur()
print(next(generateur))   # affiche : 10
print(next(generateur))   # affiche : 9
```

Un générateur grâce au mot-clef yield:

- crée un flux de données
- qui peut être sollicité au coup par coup par **next()**

Entrées/sorties - Fichiers

Lecture d'un fichier

```
fd = open("fichier", "r")
for ligne in fd :
    print(ligne, end="")
fd.close()
```

Entrées/sorties - Fichiers

Lecture d'un fichier

```
fd = open("fichier", "r")
for ligne in fd :
    print(ligne, end="")
fd.close()
```

Un fichier peut être ouvert dans différents modes. Voici les trois principaux :

- r : lecture
- w : écriture (fichier existant écrasé)
- a : écriture en ajout

Entrées/sorties - Fichiers (suite)

Transfert du contenu en mémoire centrale avec readlines()

```
fd = open("fichier", "r")
for ligne in fd.readlines() :
    print(ligne, end="")
fd.close()
```

Entrées/sorties - Fichiers (suite)

Transfert du contenu en mémoire centrale avec readlines()

```
fd = open("fichier", "r")
for ligne in fd.readlines() :
    print(ligne, end="")
fd.close()
```

Lecture ligne après ligne avec readline() ou next()

```
fd = open("fichier", "r")
print(fd.readline())
print(next(fd))
```

Entrées/sorties - Fichiers (suite)

Transfert du contenu en mémoire centrale avec readlines()

```
fd = open("fichier", "r")
for ligne in fd.readlines() :
    print(ligne, end="")
fd.close()
```

Lecture ligne après ligne avec readline() ou next()

```
fd = open("fichier", "r")
print(fd.readline())
print(next(fd))
```

next() émet une erreur à la fin du fichier

Entrées/sorties - Fichiers (suite)

Ouverture d'un fichier avec with

```
with open("fichier", "r") as fd :
    for ligne in fd :
        print(fd)
```

Entrées/sorties - Fichiers (suite)

Ouverture d'un fichier avec with

```
with open("fichier", "r") as fd :
    for ligne in fd :
        print(fd)
```

Avec with le fichier est automatiquement fermé

Entrées/sorties - Fichiers (suite)

Ecriture dans un fichier avec write() ou writelines()

```
fd = open("fichier", "w")
fd.write("Premiere␣ligne")
lignes = ["Seconde␣ligne", "Troisieme␣ligne"]
fd.writelines(lignes)
```

Entrées/sorties - Fichiers (suite)

Ecriture dans un fichier avec write() ou writelines()

```
fd = open("fichier", "w")
fd.write("Premiere␣ligne")
lignes = ["Seconde␣ligne", "Troisieme␣ligne"]
fd.writelines(lignes)
```

Problèmes d'encodage

- Précisez l'encodage lors de l'ouverture du fichier :

```
fd = open("fichier", "r", "iso-8859-15")
```

- réencodez le texte si nécessaire :

```
ligneUTF = ligne.encode("utf-8")
```

Interface Système - Accès à un exécutable

Accès au résultat d'un exécutable

```
import os
ll = os.popen("ls␣-l")
for ligne in ll :
    print(ligne)
```

Interface Système - Accès à un exécutable

Accès au résultat d'un exécutable

```
import os
ll = os.popen("ls␣-l")
for ligne in ll :
    print(ligne)
```

os.popen()

- renvoie une liste
- et donc attention à votre mémoire...

Interface Système - Accès à un exécutable

Accès au résultat d'un exécutable

```
import os
ll = os.popen("ls␣-l")
for ligne in ll :
    print(ligne)
```

os.popen()

- renvoie une liste
- et donc attention à votre mémoire...

Exécution directe d'un exécutable

```
import os
os.system("ls␣-l")
```

Interface Système - Exploration d'une arborescence de dossier

Schéma de programmation

```
import os
def parcours(repertoire) :
    print("Je␣suis␣dans␣", repertoire)
    liste = os.listdir(repertoire)
    for fichier in liste :
        if os.path.isdir(...) : ...
        if os.path.isfile(...) : ...
parcours(...)
```

Interface Système - Exploration d'une arborescence de dossier

Schéma de programmation

```
import os
def parcours(repertoire) :
    print("Je␣suis␣dans␣", repertoire)
    liste = os.listdir(repertoire)
    for fichier in liste :
        if os.path.isdir(...) : ...
        if os.path.isfile(...) : ...
parcours(...)
```

- os.listdir() : liste du répertoire
- os.path.isdir() : teste si un fichier est un dossier
- os.path.isfile() : teste si un fichier est un fichier régulier

Interface système - Test du statut d'un dossier

Pour générer des chemins avec / ou (Windows)

```
os.path.join('<chemin>','<fichier>')
```

Pour tester l'accessibilité d'un dossier :

```
os.access('<chemin/fichier>', os.X_OK)
```

Interface système - Test du statut d'un dossier

Pour générer des chemins avec / ou (Windows)

```
os.path.join('<chemin>','<fichier>')
```

Pour tester l'accessibilité d'un dossier :

```
os.access('<chemin/fichier>', os.X_OK)
```

Différents tests :

- os.F_OK : existence
- os.R_OK : droit de lecture
- os.W_OK : droit d'écriture
- os.X_OK : droit d'accès (ou d'exécution)

Analyse de données textuelles - Expressions régulières

Les expressions régulières (regexp) servent à

identifier une sous-chaîne dans une chaîne
extraire une ou plusieurs sous-chaînes dans une chaîne

Analyse de données textuelles - Expressions régulières

Les expressions régulières (regexp) servent à

identifier une sous-chaîne dans une chaîne
extraire une ou plusieurs sous-chaînes dans une chaîne

Les principales fonctions du module re

- res = re.match(regexp, chaîne) : identifie et/ou extrait des sous-chaînes en début
- res = re.search(regexp, chaîne) : identifie et/ou extrait des sous-chaînes
- liste = re.findall(regexp, chaîne) : extrait des sous-chaînes de même structure (par exemple tous les nombres)
- mots = re.split(regexp, chaîne) : découpe une chaîne suivant un séparateur
- regexpc = re.compile(regexp) : précompile l'expression régulière pour plus d'efficacité

Bestiaire des méta-caractères

- `.` : un caractère quelconque
- `*` : de 0 à n fois
- `+` : de 1 à n fois
- `?` : de 0 à 1 fois
- `{m,n}` : de m à n fois
- `[abc]` : a ou b ou c (exemple)
- `[^abc]` : ni a, ni b, ni c (exemple)
- `"^...` : motif en début de chaîne
- `...$"` : motif en fin de chaîne
- `\` : déspécialisation d'un méta-caractère
- `(...)` : extraction
  et/ou construction d'alternatives avec le pipe

Extraction des noms de pays et capitales

Afghanistan (l'),Kaboul
Bolivie (la),Sucre / La Paz

Extraction des noms de pays et capitales

Afghanistan (l'),Kaboul
Bolivie (la),Sucre / La Paz

```
import re
with open("capitales.csv") as fd :
  for ligne in fd :
    res = re.match("([^(,]+).*,([^(/]+)",ligne)
    if res :
        print(res.group(1).strip(), ":",
              res.group(2).strip())
```

Extraction des noms de pays et capitales

Afghanistan (l'),Kaboul
Bolivie (la),Sucre / La Paz

```
import re
with open("capitales.csv") as fd :
  for ligne in fd :
    res = re.match("([^(,]+).*,([^(/]+)",ligne)
    if res :
        print(res.group(1).strip(), ":",
              res.group(2).strip())
```

res.group(1) et res.group(2)

correspondent aux deux groupes de caractères extraits

Exemples d'expressions régulières

Alternatives de sous-chaînes en début

```
import re
ligne = "Madame␣Claire␣Delune"
if re.match("(Monsieur|Madame)", ligne) :
    print("Ligne␣avec␣civilite")
```

Exemples d'expressions régulières

Alternatives de sous-chaînes en début

```
import re
ligne = "Madame␣Claire␣Delune"
if re.match("(Monsieur|Madame)", ligne) :
    print("Ligne␣avec␣civilite")
```

Extractions de sous-chaînes avec déspécialisation

```
l = "...␣127.0.0.1␣-␣-␣[03/Dec/2017]␣..."
resultat = re.search("(\d+\.\d+\.\d+\.\d+)␣", l)
if resultat :
    print("Adresse␣IP␣:␣", resultat.group(1))
```

Les expressions régulières sont avides et gloutonnes

Exemple : extraction d'une extension

```
import re
test = "fichier.ext1.ext2"
resultat = re.search(".\.([^.]+)$", test)
# ou resultat = re.search(".+\.(.+)", test)
if resultat :
    print("Derniere␣extension␣=",resultat.group(1))
```

Les expressions régulières sont avides et gloutonnes

Exemple : extraction d'une extension

```
import re
test = "fichier.ext1.ext2"
resultat = re.search(".\.([^.]+)$", test)
# ou resultat = re.search(".+\.(.+)", test)
if resultat :
    print("Derniere␣extension␣=",resultat.group(1))
```

Exemple : extraction des extensions

```
resultats = re.findall("\.([^.]+)", test)
print(resultats)
```

Il est aussi possible d'utiliser re.split() voire split()

```
test = "fichier.ext1.ext2"
chaines = test.split(".")
if len(chaines) > 1 :
    print("Derniere␣extension␣=", chaines[-1])
```

Il est aussi possible d'utiliser re.split() voire split()

```
test = "fichier.ext1.ext2"
chaines = test.split(".")
if len(chaines) > 1 :
    print("Derniere␣extension␣=", chaines[-1])
```

chaines

```
["fichier", "ext1", "ext2"]
```

Bestiaire des méta-caractères (suite)

- `\d` : équivalent à `[0-9]`
- `\D` : équivalent à `[^0-9]`
- `\w` : équivalent à `[_a-zA-Z0-9]`
- `\W` : équivalent à `[^_a-zA-Z0-9]`
- `\s` : équivalent à `[ \n\r\t\f]`
- `\S` : équivalent à `[^ \n\r\t\f]`

Entrées/sorties - Gestion des exceptions

Le traitement des exceptions permet de reprendre la main en cas d'une erreur

qui sinon arrêterait l'exécution du script

Entrées/sorties - Gestion des exceptions

Le traitement des exceptions permet de reprendre la main en cas d'une erreur

qui sinon arrêterait l'exécution du script

Il repose sur trois clauses principales

- try : bloc d'instructions "protégé"
- except : bloc d'instructions à exécuter en cas d'erreurs
- else : bloc d'instructions positionné après les clauses exec

Entrées/sorties - Gestion des exceptions (suite)

Exemple de traitement des exceptions

```
try:
    fd = open("JeNExistePas", "r")
    #valeur = int("abc")

except OSError as err:
    print("OS␣error:␣{}".format(err))
    print("suite␣en␣cas␣d'erreur␣systeme")
except ValueError as err:
    print("Value␣Error:␣{}".format(err))
    print("suite␣sur␣une␣erreur␣de␣valeur")

else :
    print("Nb␣lignes␣=", len(fd.readlines()))
    fd.close()
```

Programmation orientée objet

Une classe définit une spécification formelle d'un concept qui réunit :

- les informations définissant un concept
- avec les traitements pouvant être effectués sur ces données (les méthodes)

Programmation orientée objet

Une classe définit une spécification formelle d'un concept qui réunit :

- les informations définissant un concept
- avec les traitements pouvant être effectués sur ces données (les méthodes)

Une classe est instanciable en un objet

L'objet est une "incarnation" du concept

Programmation orientée objet

Une classe définit une spécification formelle d'un concept qui réunit :

- les informations définissant un concept
- avec les traitements pouvant être effectués sur ces données (les méthodes)

Une classe est instanciable en un objet

L'objet est une "incarnation" du concept

Un objet est un segment de mémoire

Le nom de l'objet est une adresse qui permet d'accéder à ce segment de mémoire

Une classe implémentant le concept de Champignon

```
class Champignon :
    nomScientifique = "Fungi"

    def __init__(self, espece, comestibilite) :
        self.espece = espece
        self.comestibilite = comestibilite

print(Champignon.nomScientifique) # Fungi
amanite = Champignon("des␣Cesars", "excellente")
print(amanite.espece)  # des Cesars
print(amanite.comestibilite)  # excellente
```

Programmation orientée objet

Une classe implémentant le concept de Champignon

```
class Champignon :
    nomScientifique = "Fungi"

    def __init__(self, espece, comestibilite) :
        self.espece = espece
        self.comestibilite = comestibilite

print(Champignon.nomScientifique) # Fungi
amanite = Champignon("des␣Cesars", "excellente")
print(amanite.espece)  # des Cesars
print(amanite.comestibilite)  # excellente
```

Le paramètre **self** en première position dans les méthodes

représente l'objet qui utilisera cette classe pour exister